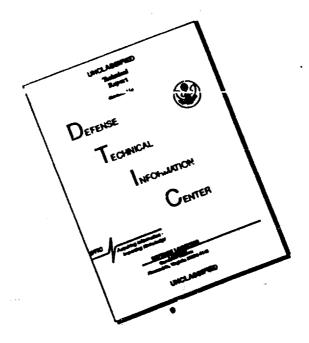
LOAN DOCUMENT

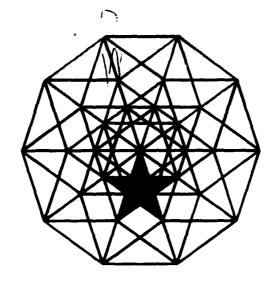
10			РНОТО	OGRAPH THIS SHEET	(D)	
195	UMBER	LEVEL]		INVENTORY	-
AD-A268	DTIC ACCESSION NUMBER	F05R-	DISTRIBUTION Approved to	91	j.	H A N D
		<u> </u>		DISTRIBUTION STATE	MENT	L
ACCESSION FOR NTIS GRAAI DTIC TRAC UNANNOUNCED JUSTIFICATION BY DISTRIBUTION/ AVAILABILITY CODES DISTRIBUTION AVAILABILITY	Z ANDAOR SPECIAL			S	DTIC ELECTF AUG 9 1993 C	E W I T H
P-1					DATE ACCESSIONED	
				<u></u>		\int_{Λ}
DISTRIBUTION						A R
DTIC QU	ALITY INSP	ECTED 3				E
			···		DATE RETURNED	1
				93-18	3 1 0 1 	
	DATE RECE	IVED IN DTIC		REC	GISTERED OR CERTIFIED NUMBER	1
	PI	HOTOGRAPH T	THIS SHEET AND RETU	RN TO DTIC-FDAC		
DTIC KIM 70A			DOCUMENT PROCESSING LOAN DOCUME		MENOR EDIALITED	Ą

93805 154

JISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.



The Fifth Distributed Memory Computing Conference

April 8-12, 1990

Charleston, South Carolina

Proceedings Volume I **Applications**

Edited by:

David W. Walker and Quentin F. Stout

Host Institution:
The University of South Carolina



REPORT DOCUMENTATION PAGE

m Approved OMB No 3704-0188

Public reporting purgen for this in among it information is estimated to surrice in our deriresponse including the time for reviewing instructions, searching existing data source

gathering and maintaining the data herderd, and combletting and reviewing the literation of information abend comments regal collection of information, including suggestions for reducing this burden to Mashington eadquarters Services, Directorate for Davis Highway, Suite 1204. Arlington, via. 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Personal Conference of the Conference	rding this burden estimate or any other aspect of this rinformation Operations and Reports (1215) Jefferson ect. 0704(0188), Washington (10) 40503
1. AGENCY USE ONLY (Leave blank) 2. REPORT DATE 3. REPORT TYPE AND FINAL OF AF	2R 26 31 MAR 91
4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
THE FIFTH DISTRIBUTED MEMORY COMPUTING CONFERENCE VOL I	AFOSR-90-0212
6. AUTHOR(S)	
DR. DAVID WALKER	61102F 2304/A3
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS.ES; UNIVERSITY OF SOUTH CAROLINA COLUMBIA. SC 29208	3. PERFORMING ORGANIZATION REPORT NUMBER
	R. 93 049 0
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFOSR/NE Bldg 410 Bolling AFB DC 20532-6448	AFOSR-90-0212
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution unlimited.	
13. ABSTRACT (Maximum 200 words)	

Controlling interplanetary spacecraft and planning their activities, as currently practiced, requires massive amounts of computer time and personnel. To improve this situation, it is desired to use advanced computing to speed up and automate the commanding process. Several design and prototype efforts have been underway at JPL to understand the appropriate roles for concurrent processors in future interplanetary spacecraft operations. Here we report on an effort to identify likely candidates for parallelism among existing software systems that both generate commands to be sent to the spacecraft and simulate what the spacecraft will do with these commands when it receives them. We also describe promising results form effort to create parallel prototype of representative portions of these systems

14. SUBJECT TERMS			15. NUMBER OF PAGES	
			13. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICAT ON OF THIS PAGE	SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABS	TRACT
INCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR	i

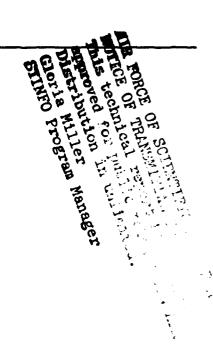
on the JPL/Caltech Mark III hypercube.

AEOSR-TR- 93 0490

Proceedings of Approved &

The Fifth Distributed Memory Computing Conference

Volume I **Applications**



Proceedings of

The Fifth **Distributed Memory Computing Conference**

April 8-12, 1990

Charleston, South Carolina

Volume I **Applications**

Edited by:

David W. Walker University of South Carolina

and

Quentin F. Stout University of Michigan

Host Institution: The University of South Carolina



IEEE Computer Society Press Los Alamitos, California

Washington ● Brussels ●

Tokyo

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society Press, or The Institute of Electrical and Electronics Engineers, Inc.

Published by



IEEE Computer Society Press 10662 Los Vaqueros Circle P.O. Box 3014 Los Alamitos, CA 90720-1264

Copyright © 1990 by the Institute of Electrical and Electronics Engineers, Inc.

Cover by Wally Hutchins

Printed in United States of America

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 29 Congress Street, Salem, MA 01970. Instructors are permitted to photocopy isolated articles for noncommercial classroom use without fee. For other copying, reprint or republication permission, write to Director, Publishing Services, IEEE, 345 East 47th Street, New York, NY 10017. All rights reserved.

IEEE Computer Society Press Order Number 2113 Library of Congress Number 90-82988 ISBN 0-8186-2113-3 (paper) ISBN 0-8186-6089-9 (microfiche) SAN 264-620X

Additional copies can be ordered from:

IEEE Computer Society Press Customer Service Center 10662 Los Vaqueros Circle P.O. Box 3014 Los Alamitos, CA 90720-1264 IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM

IEEE Computer Society
Coshima Building
2-19-1 Minami-Aoyama,
Minato-Ku
Tokyo 107, JAPAN

IEEE Service Center 445 Hoes Lane P.O. Box 1331 Piscataway, NJ 08855-1331



The Institute of Electrical and Electronics Engineers, Inc.

Preface

The Fifth Distributed Memory Computing Conference (DMCC5) was held April 8-12, 1990, at The Omni Hotel, Charleston, South Carolina, and was hosted by the University of South Carolina. Four invited talks and 99 contributed talks were presented, with 12 more papers making up the two mini-symposia. In addition, approximately 100 posters were presented. This two-volume set includes papers from all four of these categories. Volume 1 covers applications, and Volume 2 deals with all other areas, including hardware, software tools, performance, languages, and so on.

DMCC5 continues the conference series, previously known as the "Hypercube" or "HCCA" conference, that originated in 1985 at the Oak Ridge National Laboratory (ORNL). The first two conferences were hosted by ORNL in Knoxville, TN, and focused almost exclusively on the hypercube concurrent computer. The scope of the third and fourth conferences, respectively hosted by Caltech's Jet Propulsion Laboratory, and Sandia National Laboratories, was broadened to include other types of distributed memory computers. With DMCC5 this trend has continued, and as the new name indicates, the conference series now embraces all aspects of distributed memory computing.

The DMCC5 conference theme was "Education", which we believe is essential in encouraging the effective use of distributed memory computers. This theme was promoted by half-day tutorials, student conference awards, and a student paper competition. A grant from the National Science Foundation provided funds for 26 student attendees, many of whom might otherwise have been unable to participate in the conference. The Student Paper Competition (for papers authored solely by students) generated several good entries. IBM Corporation generously sponsored 3 first prizes of \$500 each. Three runner-up prizes were sponsored by the College of Science and Mathematics of the University of South Carolina, and an additional runner-up prize was donated by the Caltech Concurrent Computation Program. Prize winners are listed on the following page. We are grateful to the NSF, and the sponsors of the Student Paper Competition, for their support of student participation in DMCC5.

A large number of persons and organizations have contributed to the success of DMCC5. We are particularly grateful to the conference sponsors:

Air Force Office of Scientific Research
Defense Advanced Research Projects Agency, ISTO
Joint Tactical Fusion Program Office
NASA Ames Research Laboratory
Sandia National Laboratories
Strategic Defense Initiative Organization/OIST
U.S. Air Force, Electronic Systems Division

We would also like to thank the members of the Organizing and Program Committees for ensuring the smooth running of the conference. Also essential to the conference organization were those who gave their time and expertise to serve as session chairs, and reviewers, and participants in the panel discussion. Finally, we are grateful for the support of the DMCC5 host institution, the University of South Carolina.

David W. Walker Quentin F. Stout

Student Paper Competition Awards

First Prize (Hardware)

Philip R. Miller and Jelio T. Yantchev, Department of Electronics and Computer Science, University of Southampton, UK, "Developing Powerful Communication Mechanisms for Distributed Memory Computers from Simple and Efficient Message Routing."

First Prize (Algorithms and Applications)

Stefan Vandewalle, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, "Waveform Relaxation Methods for Solving Parabolic Partial Differential Equations."

First Prize (Operating Systems and Software)

Anthony Skjellum and Alvin P. Leung, Department of Chemical Engineering, California Institute of Technology, "Zipcode: A Portable Multicomputer Communication Library Atop the Reactive Kernel."

Runner-Up Prizes

Anne C. Elster, School of Electrical Engineering, Cornell University, "Basic Matrix Subprograms for Distributed Memory Systems."

Arjun Khanna, Department of Electrical and Computer Engineering, University of Texas at Austin, "On Managing Classes in a Distributed Object-Oriented Operating System."

Silvia M. Muller, Department of Computer Science, University of Saarland, West Germany, "A Method to Parallelize Tridiagonal Solvers."

Anthony Skjellum and Alvin P. Leung, Department of Chemical Engineering, California Institute of Technology, "LU Factorization of Sparse, Unsymmetric Jacobian Matrices on Multicomputers: Experience, Strategies, Performance."

Organizing Committee

Paul G. Huray, General Chairman University of South Carolina

Quentin F. Stout, Program Co-Chairman University of Michigan

David W. Walker, Program Co-Chairman University of South Carolina

Terrance L. Huntsberger, Vendor Exhibit Chairman University of South Carolina

Jane W. Squires, Conference Administrator University of South Carolina

Donald M. Austin
U. S. Department of Energy

F. Ron Bailey
NASA Ames Research Center

Robert E. Benner Sandia National Laboratories

Sudhir Bhagwan
Oregon Advanced Computing Institute

Terry Cole
Jet Propulsion Laboratory

Geoffrey C. Fox California Institute of Technology

Michael T. Heath Oak Ridge National Laboratory

Anthony J. G. Hey University of Southampton

S. Lennart Johnsson Yale University

Paul Messina California Institute of Technology

C. Edward Oliver Oak Ridge National Laboratory

Horst D. Simon NASA Ames Research Center

Stephen L. Squires
Defense Advanced Research Projects Agency, ISTO

Gilbert G. Weigand Sandia National Laboratories

Pat Windham U. S. Senate CST Committee

Program Committee

Quentin F. Stout, Program Co-Chairman University of Michigan

David W. Walker, Program Co-Chairman University of South Carolina

William Athas
University of Texas at Austin

Robert E. Benner Sandia National Laboratories

Geoffrey C. Fox California Institute of Technology

Joydeep Ghosh University of Texas at Austin

John L. Gustafson Ames Laboratory - USDOE, Iowa State University

John P. Hayes University of Michigan

Terrance L. Huntsberger University of South Carolina

Ted Lewis
Oregon Advanced Computing Institute

Paul Messina California Institute of Technology

Steve W. Otto California Institute of Technology

Daniel A. Reed University of Illinois

Paul Reynolds University of Virginia

P. Sadayappan
The Ohio State University

Horst D. Simon NASA Ames Research Center

Robert C. Ward Oak Ridge National Laboratory

Patrick H. Worley
Oak Ridge National Laboratory

Steering Committee

Michael T. Heath, Chairman Oak Ridge National Laboratory

Terry Cole Jet Propulsion Laboratory

John L. Gustafson Ames Laboratory – USDOE, Iowa State University

Paul Messina California Institute of Technology

Gilbert G. Weigand Sandia National Laboratories

Organizational Assistance

Leigh Hayes University of South Carolina

Donna ReynoldsPalmetto Economic Development Corporation

Courtnay Squires
University of South Carolina

Session Chairs

William Athas
Michael Barton

Edward A. Carmona

Alva L. Couch

David Curkendall Robert D. Ferraro

Paul Frederickson W. Kent Fuchs

Dirk Grunwald

John L. Gustafson

Michael T. Heath Anthony J. G. Hey

Ching-Tien Ho

Leendert M. Huisman S. Lennart Johnsson

Judson P. Jones

Ted Lewis

Manton M. Matthews

Paul Messina

Russ Miller Arthur A. Mirin

C. Edward Oliver

Steve W. Otto Roy P. Pargas

John L. Pfaltz Walter Rudd

P. Sadayappan

Joel Saltz

Steven R. Seidel

Richard Sincovec

Quentin F. Stout

Robert C. Ward Roy D. Williams

Mike Wolfe

Patrick H. Worley

University of Texas at Austin

Intel Scientific Computers

Air Force Weapons Laboratory

Tufts University

Jet Propulsion Laboratory

Jet Propulsion Laboratory

RIACS, NASA Ames Research Center University of Illinois at Urbana-Champaign

University of Colorado at Boulder

Ames Laboratory - USDOE, Iowa State University

Oak Ridge National Laboratory University of Southampton

IBM Almaden Research Center

IBM Thomas J. Watson Research Center Thinking Machines Corp. and Yale University

Oak Ridge National Laboratory

Oregon Advanced Computing Institute

University of South Carolina

California Institute of Technology

State University of New York at Buffalo Lawrence Livermore National Laboratory

Oak Ridge National Laboratory
California Institute of Technology

Clemson University

University of Virginia
Oregon State University

Ohio State University

ICASE, NASA Langley Research Center

Michigan Technological University

RIACS, NASA Ames Research Center

University of Michigan

Oak Ridge National Laboratory California Institute of Technology

Oregon Graduate Institute

Oak Ridge National Laboratory

Reviewers

In addition to those already listed as members of the Program Committee and Session Chairs, the following people assisted in reviewing papers.

Marsha Berger
Sungwoon Choi
Oregon State University
Raymond Cline
Sandia National Laboratories

William Dally Massachusetts Institute of Technology

Wayne Davidson Cogent Research Inc.
Phil Dickens University of Virginia

Joan Francioni North Carolina Supercomputer Center

Richard Fujimoto Georgia Institute of Technology

Gary Garunke Sequent Computer Corp.
Andrew Grimshaw University of Virginia

Rajiv Gupta North American Philips Corporation

Susanne Hambrusch Purdue University

William Harrison University of Illinois at Urbana-Champaign Ralph Johnson University of Illinois at Urbana-Champaign

Jeffrey N. Jortner Sandia National Laboratories

Ken Kennedy Rice University

Inkyu Ki.n

Oregon State University

Ten-Hwang Lai

Chio State University

Parasoft Corporation

Jim Li University of Illinois at Urbana-Champaign

Richard Ma

Worthy Martin

Piyush Mehrota

Trevor Mudge

Lionel Ni

David Nicol

Aerospace Corporation

University of Virginia

Purdue University

University of Michigan

Michigan State University

College of William and Mary

Kathleen Nichols Apple Computer Inc.

David Notkin University of Washington, Seattle

David Padua University of Illinois at Urbana-Champaign
Ben Peek Oregon Advanced Computing Institute

Mike Quinn Oregon State University

C. S. Raghavendra
University of Southern California
Georgia Institute of Technology

Bob Rau Hewlett-Packard Laboratories, Palo Alto Tony Reeves Cornell University Larry Snyder University of Washington, Seattle **David Socha** University of Washington, Seattle J. A. Stankovic University of Massachusetts at Amhurst **Binay Sugla** AT&T Bell Laboratories Ewan Tempero University of Washington, Seattle **Brian Totty** University of Illinois at Urbana-Champaign Martin Waugh Intel Scientific Computers Craig Williams University of Virginia Larry Wittie State University of New York at Stony Brook Kun-Lung Wu University of Illinois at Urbana-Champaign

Panel Discussion

We are grateful to the following people for participating in the panel discussion "Is Massive Parallelism Ready for the Masses?", which was chaired and organized by Horst D. Simon.

Horst D. Simon

NASA Ames Research Center

Greg Astfalk

Convex Computer Corporation

Richard Clayton

Thinking Machines Corporation

John L. Gustafson Ames Laboratory - USDOE, Iowa State University

George Michael Lawrence Livermore National Laboratory

Hossur Srikantan Union Bank of Switserland Christopher G. Willard Dataquest

Table of Contents

Preface
Organizational Assistance
Panel Discussion
Volume I
Session 1: Expert Systems
Hypercubes for Critical Space Flight Command Operations
Session 2: Alternate Applications
Parallel Distributed-Memory Implementation of the Corrective Switching Problem 30 JY. Blanc, D. Trystram, and J.W.A. Ryckbosch Fault Simulation on Message Passing Parallel Processors
Problems for Distributed Architectures
Concurrent Implementation of Munkres Algorithm
Multi-Tiered Algorithms for 2-Dimensional Bin Packing
Efficient Serial and Parallel Subcube Recognition in Hypercubes
Parallel Thinning on a Distributed Memory Machine
Session 3: Multi-Target Tracking
A Nonconvex Cost Optimization Approach to Tracking Multiple Targets by a Parallel Computation Network

Session 4: Simulation of Systems and Discrete Events

Parallel Discrete Event Simulation Using Synchronized Event Schedulers	0
Air Traffic Simulation: An Object Oriented, Discrete Event Simulation on the Intel iPSC/2 Parallel System	5
W.L. Bain	
Application of Transputers to Aircraft Simulation and Control	1
Simulation of an Urban Mobile Radio Channel on the Myrias SPS-2	7
M. Fattouche, L. Petherick, and A. Fapojuwo	
Portable Asteroids on Hypercube or Transputers	.1
A General Framework for Complex Time-Driven Simulations on Hypercubes	.7
D.L. Meier, K.L. Cloud, J.C. Horvath,	
L.D. Allan, W.H. Hammond, and H.A. Maxfield	
Session 5: Path Planning and Navigation	
Path Planning on a Distributed Memory Computer	4
S. Miguet and Y. Robert	
Learning to Plan Near-Optimal Collision-Free Paths) [
Parallel Algorithms for One- and Two-Vehicle Navigation	ŀO
E. Gurewitz, G.C. Fox, and YF. Wong	
A Neural Network Approach to Multi-Vehicle Navigation	8
G.C. Fox, E. Gurewitz, and YF. Wong	
Session 6: Data and Image Processing	
A Connectionist Technique for Data Smoothing	14
R. Daniel, Jr. and K. Teague	•^
Component Labeling Algorithms on an Intel iPSC/2 Hypercube	פנ
Digital Halftoning by Parallel Simulation of Neural Networks	35
R.M. Geist, R.P. Pargas, and P.K. Khambekar	_
Hypercube Algorithms for Image Decomposition and Analysis in the	
Wavelet Representation	/1
T.L. Huntsberger and B.A. Huntsberger	
Parallel Processing Applied to 3D Coronary Arteriography	16
A. Sarwal, J. Ramanathan, D.L. Parker, and J. Wu	
Session 7: Computer Vision	
Surface Reconstruction and Discontinuity Detection: A Fast	
Hierarchical Approach on a Two-Dimensional Mesh	34
D D_mili	
R. Battiti	١.
R. Battiti An Adaptive Multiscale Scheme for Real-Time Motion Field Estimation)4
An Adaptive Multiscale Scheme for Real-Time Motion Field Estimation	94
An Adaptive Multiscale Scheme for Real-Time Motion Field Estimation	

The Hypercube Ray Tracer
Session 9: Sorting
Parallel Sorting on Symult 2010
Parallel Sorting on the Hypercube Concurrent Processor
Session 10: Mathematical Methods
Parallel Methods for Solving Polynomial Problems on
Distributed Memory Multicomputers
Applications of Adaptive Data Distributions
The Quadratic Sieve Factoring Algorithm on Distributed Memory Multiprocessors 254 M. Cosnard and JL. Philippe
Parallel Quasi-Newton Methods for Unconstrained Optimization
Parallel Nonlinear Optimization
Parallelizing Multiple Linear Regression for Speed and Redundancy: An Empirical Study
Session 11: Full and Banded Matrix Algorithms
Solving Very Large Dense Systems of Linear Equations on the iPSC/860 286 D.S. Scott, E. Castro-Leon, and E.J. Kushner
Parallel Solution Algorithms for the Triangular Sylvester Equation
Reducing Inner Product Computation in the Parallel One-Sided Jacobi Algorithm 301 C. Romine and K. Sigmon
Basic Matrix Subprograms for Distributed Memory Systems
Linear Algebra for Dense Matrices on a Hypercube
Session 12: Sparse Matrix Algorithms
Incremental Condition Estimator for Parallel Sparse Matrix Factorization
Experience, Strategies, Performance

Session 13: Indiagonal systems
A Method to Parallelize Tridiagonal Solvers
Solution of Periodic Tridiagonal Linear Systems on a Hypercube
The Error Analysis of a Tridiagonal Solver
Session 14: Basic Algorithms
An Efficient FFT Algorithm on Multiprocessors with Distributed Memory
Distributed Evaluation of an Iterative Function for All Object Pairs on an SIMD Hypercube
The Complexity of Reshaping Arrays on Boolean Cubes
Random Number Generation in the Parallel Environment
Session 15: Monte Carlo Physics
Cluster Algorithms for Spin Models on MIMD Parallel Computers
Quantum Spin Calculations on a Hypercube Parallel Supercomputer
Lattice QCD: Commercial vs. Home-Grown Parallel Computers
Session 16: Electromagnetic Scattering Problems
The Finite Element Solution of Two-Dimensional Transverse Magnetic Scattering Problems on the Connection Machine
S. Hutchinson, S. Castillo, E. Hensel, and K. Dalton Parallel Finite Elements Applied to the Electromagnetic Scattering Problem
R.D. Ferraro, T. Cwik, N. Jacobi, P.C. Liewer, T.G. Lockhart, G.A. Lyzenga, J. Parker, and J.E. Patterson
An Examination of Finite Element Formulations and Parameters for Accurate Parallel Solution of Electromagnetic Scattering Problems
J.W. Parker, R.D. Ferraro, and P.C. Liewer
Session 17: Plasma Physics Applications
Massively Parallel Fokker-Planck Calculations
Implementing Particle-in-Cell Plasma Simulation Code on the BBN TC2000
A 2D Electrostatic PIC Code for the Mark III Hypercube

Session 18: Computational Fluid Dynamics

Massively Parallel Computation of the Euler Equations	6
Concurrent Implementation of a Fast Vortex Method	3
F. Pépin and A. Leonard Parallel Computation of the Compressible Navier-Stokes Equations with a	
Pressure-Correction Algorithm	. 2
M.E. Braaten	U
Session 19: Other Scientific Applications	
Hypercube Simulation of Electric Fish Potentials	0
R. Williams, B. Rasnow, and C. Assad	
Molecular Dynamics Simulations of Short-Range Force Systems on	
1024-Node Hypercubes	
Transputer Modelling of Be Star Circumstellar Discs	
A Hypercube Application in Large Scale Composite Materials Modeling	
Electron-Molecule Collisions on the Mark IIIfp Hypercube	8
Modeling High-Temperature Superconductors and	
Metallic Alloys on the Intel iPSC/860	4
Parallel Solutions to the Phase Problem in X-Ray Crystallography	.3
An Automata Model of Granular Materials	2
Seismic Modeling and Inversion on the NCUBE	0
J. Sochacki, P. O'Leary, C. Bennett, R.E. Ewing, and R.C. Sharpley	
Session 20: Structural Analysis	
Implementation of JAC3D on the NCUBE/ten	8
Porting the ABAQUS Structural Analysis Code to Run on the iPSC/254	5
M.L. Barton and E.J. Kushner	
Session 21: PDE Methods	
Conjugate Gradient Methods for Spline Collocation Equations	i 0
Multigrid on Massively Parallel Computers	9
A Parallel Algorithm for Solving Higher KdV Equations on a Hypercube	4
The Triangle Method for Saving Startup Time in Parallel Computers	8

Session 22: Mini-Symposium on Concurrent Simulation Paradigms

Waveform Relaxation Methods for Solving Parabolic Partial Differential Equations 575
S. Vandewalle
A Parallel Implementation of ESACAP
S. Skelboe
Concurrent DASSL Applied to Dynamic Distillation Column Simulation
A. Skjellum and M. Morari
Convergence and Circuit Partitioning Aspects for Waveform Relaxation
Ū. Miekkala, O. Nevanlinna, and A. Ruehli
Partitioning Tradeoffs for Waveform Relaxation in Transient
Analysis Circuit Simulation
L. Peterson and S. Mattisson
Distributed Model Evaluation for the Waveform Relaxation Method
L. Olsson, L. Peterson, and S. Mattisson

Volume II

Session 23: Overviews
Concurrent Supercomputing in Europe
Touchstone Program Overview
Session 24: Dual Ported Memory Computers
Communication on H16: A Study of Methods and Performance in a Hypercubic Network Based on Dual Port RAM
Session 25: Shared Memory
Design and Implementation of a Multi-Cache System on a
Loosely Coupled Multiprocessor
Hot-Spot Performance of Single-Stage and Multistage Interconnection Networks
Programming the PLUS Distributed-Memory System
Parallel Processor Memory Reference Analysis and its Application to Interconnect Architecture
Session 26: Other Hardware and Architectures
A Heterogeneous Hypercube Based on Strengthened Nodes for a Fast Processing of SAR Raw-Data
An SIMD Multiprocessor Using DSP Microprocessors
A Reconfigurable Reduced-Bus Multiprocessor Interconnection Network 719 T. Ramesh and S. Ganesan
An Orthogonal Multiprocessor with Snooping Caches
Session 27: Distributed Computing
DAWGS: A Distributed Compute Server Utilizing Idle Workstations
HIGHLAND: A Graph-Based Parallel Processing Environment for Heterogeneous Local Area Networks
Session 28: Communication Systems
A Deadlock-Free Communicating Kernel for Binary N-Cube Architectures

Mapping and Compiled Communication on the Connection Machine System
Zipcode: A Portable Multicomputer Communication Library Atop the
Reactive Kernel
A. Skjellum and A.P. Leung Desynchronized Communication Schemes on Distributed-Memory Architectures
JY. Blanc, D. Trystram, and G. Villard
MMPS: Portable Message Passing Support for Parallel Computing
The Performance/Functionality Dilemma of Multicomputer Message Passing
Extension of the iPSC/2 Message Passing System with the
Select-by-Sender Functionality
Session 29: Routing
Optimal Self-Routing of Linear-Complement Permutations in Hypercubes 800 R. Boppana and C.S. Raghavendra
Developing Powerful Communication Mechanisms for Distributed
Memory Computers from Simple and Efficient Message Routing
P.R. Miller and J.T. Yantchev Routing Frequently Used Bijections on Hypercube
K. Zemoudeh and A. Sengupta
Session 30: Fault Tolerance
Distributed Fault-Tolerant Embeddings of Rings in Hypercubes
Shortest Path Routing in a Failsoft Hypercube Database Machine
An Approach to Reconfigure a Fault-Tolerant Loop System
Fault Tolerant Computing: An Improved Recursive Algorithm
Session 31: Matrix Decomposition and Aliocation
An Efficient Method for Distributing Data in Hypercube Computers
An Algorithm Producing Balanced Partitionings of Data Arrays
An Input/Output Algorithm for M-Dimensional Rectangular Domain Decompositions on N-Dimensional Hypercube Multicomputers
Session 32: Data Allocation and Mapping
Mapping Data to Processors in Distributed Memory Computations
Resource Allocation in Hypercube Systems
GM. Chiu and C.S. Raghavendra A Class of Mapping Algorithms for Hypercube Computers
Y. Moon and J. Sklansky A Task Mapping Method for a Hypercube by Combining Subcubes
S. Horiike

An Empirical Study of Data Partitioning and Replication in Parallel Simulation 915 $F.$ Wieland, L. Hawley, and L. Blume
On Distributing Linked Lists
Session 33: Dynamic Load Balancing for Spatial Domains
Recursive Partitions on Multicomputers
Dynamic Load Balancing in a Concurrent Plasma PIC Code on the JPL/Caltech Mark III Hypercube
Hierarchical Domain Decomposition with Unitary Load Balancing for Electromagnetic Particle-in-Cell Codes
A Run-Time Load Balancing Strategy for Highly Parallel Systems
Hypercube Dynamic Load Balancing
Session 34: Load Distribution
Experimental Comparison of Bidding and Drafting Load Sharing Protocols
Scheduling Real-Time Computations on Hypercubes with Load Balancing 978 KJ. Lin, JY. Chung, and J.WS. Liu Empirical Comparison of Heuristic Load Distribution in
Point-to-Point Multicomputer Networks
Parallel Processing of Medium-Grain Tasks
A Hierarchical Approach to Load Balancing in Distributed Systems
Session 35: Data Parallel Programming
Scalable Abstractions for Parallel Programming
Distributed Memory Computers
A Scheme for Supporting Automatic Data Migration on Multicomputers
Session 36: Object Oriented Programming
Experience with Concurrent Aggregates (CA): Implementation and Programming 1040 A.A. Chien and W.J. Dally
Aggregate Distributed Objects for Distributed Memory Parallel Systems
On Managing Classes in a Distributed Object-Oriented Operating System

The Mentat Run-Time System: Support for Medium Grain Parallel Computation
A Concurrent Object-Oriented Programming Language and Its Distributed Implementation
Session 37: Automatic Exploitation of Parallelism
Architectural Support for Efficient Execution of Reusable Software Components
Nested Loop Tiling for Distributed Memory Machines
Parallel Loops on Distributed Machines
Memory Parallel Computers
Data Distribution in Pandore
Session 38: Parallel Languages
APL on the DATIS-P Parallel Machine
A Systolic Array Programming Language
A Distributed Memory Implementation of SISAL
Experiences with Bilingual Parallel Programming
E.T. Ong, K.M. George, and K.A. Teague
Session 39: Software Development Tools
Parallel Programs as X-Window Clients: An Implementation
An Interactive Environment for Data Partitioning and Distribution
Task Grapher: A Tool for Scheduling Parallel Program Tasks
Session 40: Performance Monitoring and Profiling
Instrumentation and Performance Monitoring of Distributed Systems
Monitoring Parallel Executions in Real Time
Design of a Communication Modeling Tool for Debugging Parallel Programs 1197 J.M. Francioni and M. Gach
Visualizing the Performance of Parallel Matrix Algorithms
Visual Animation of Paralel Algorithms for Matrix Computations

Visualization: An Aid to Design and Understand Neural Networks in a
Parallel Environment
Pictures of Performance: Highlighting Program Activity in Time and Space
Session 41: Performance Evaluation and Analysis
Performance Results on the Intel Touchstone Gamma Prototype
Parallel Computers with 100% Speedup
Fixed Time, Tiered Memory, and Superlinear Speedup
An Empirical Analysis of Parallelization Decisions Affecting
Parallel Simulation Performance
Emulation Through Time Dilation
Performance Characterization of ES-Kit Distributed Environments
Distributed Algorithms for Multi-Channel Broadcast Networks
The Impact of Sparsity and Mapping on a Concurrent Finite Element Code
The 600 Megaflops Performance of the QCD Code on the Mark IIIfp Hypercube
Synchronized Blocking in a Distributed Memory System
Performance of Mutual Exclusion Algorithms on Hypercubes
Session 42: Communication Performance
Performance Evaluation of Multicomputer Networks for Real-Time Computing
Complexity of the Symmetric Matrix-Vector Product on a Ring of Processors 1324 K. Grigg, S. Miguet, and Y. Robert
Refining the Communication Model for the Intel iPSC/2
Complexity of Scattering on a Ring of Processors
Parallel Implementation of Triangular Computation Graphs on the iPSC/2 ad the iPSC/860
J.E. Brandenburg and D.S. Scott
Communication Parameter Tests and Parallel Back Propogation Algorithms on the iPSC/2 Hypercube Multiprocessor
B.K. Mak and Ö. Egecioglu
Session 43: Embeddings
Embedding Meshes into Small Boolean Cubes

Local Search Variants for Hypercube Embedding
Multiple Network Embeddings into Hypercubes
A.K. Gupta and S.E. Hambrusch Embedding a Pyramid on the Hypercube with Minimal Routing Load 1393 R.K. Sen
Session 44: Database and File Systems
A Large Scale File Processing Application on a Hypercube
The Design and Analysis of a Tightly Coupled Hypercube File System
Analysis of Distributed Join Algorithms
Session 45: Education
Tranferring Parallel Processing Technology to Undergraduate Computer Science Students
Session 46: Minisymposium on Fault Tolerance in Real-Time Distributed Memory Computing
Quick Recovery of Embedded Structures in Hypercube Computers
A Method for Evaluating Message Communication in Faulty Hypercubes 1436 M. Peercy and P. Banerjee
Graceful Degradation on Hypercube Multiprocessors
Using Data Redistribution
Embeddings, Communication and Performance of
Algorithms in Faulty Hypercubes
Load Sharing in Hypercube Multicomputers in the Presence of Node Failures

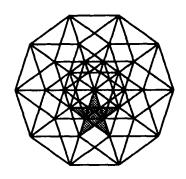
Author Index

AL-11 D 000	Chin C M
Abali, B	Chiu, GM
Aboelaze, M	Christara, C.C
Al-Bassam, S	Chung, JY
Allan, L.D	Clark, H
Aloisio, G	
André, F	Cloud, K.L
Angus, I.G	Coddington, P.D
Antonishek, J.K	Coe, M.J
Aske, O.J	Cole, R.C
Assad, C	Cosnard, M
Baek, J.H	Couch, A.L
Bailey, D.H	Crovella, M
Baillie, C.F	Cwik, T
Bain, W.L	Dahl, E.D
Balasundaram, V	Dally, W.J
Baldwin, C.H	Dalton, K
Bandyopadhyay, A.K	Daniel, Jr., R
Banerjee, P	Daoud, R.B
Barlow, J.L	Das, P.K
Barszcz, E	Davis, IV, N.J
Barton, M.L	Dawson, J.M
Bashir, N	Decyk, V.K
Bataineh, A	DeTitta, G
Battiti, R	Dharmaraj, S
Beard, R.A	Ding, HQ
Beard, R.A	Ding, HQ
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530	Ding, HQ
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. .1236
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. .1236 Fattouche, M. .107
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. .1236 Fattouche, M. .107 Fenrich, R. .58
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carmona, E.A. 943	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. 568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. 794, 876 Erçal, F. 364 Ewing, R.E. 530 Falsafi, B. 159 Fapojuwo, A. 107 Fatoohi, R.A. 1236 Fattouche, M. 107 Fenrich, R. .58 Ferraro, R.D. 417, 421, 440
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carmona, E.A. 943 Carter, M.B. 212, 217	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. 568 El-Rewini, H. 64, 1171 Elster, A.C. 311 Embrechts, H. 794, 876 Erçal, F. 364 Ewing, R.E. 530 Falsafi, B. 159 Fapojuwo, A. 107 Fatoohi, R.A. 1236 Fattouche, M. 107 Fenrich, R. 58 Ferraro, R.D. 417, 421, 440 Flower, J.W. 1105
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carmona, E.A. 943 Carter, M.B. 212, 217 Castillo, S. 408	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. 794, 876 Erçal, F. 364 Ewing, R.E. 530 Falsafi, B. 159 Fapojuwo, A. 107 Fattouche, M. 107 Fenrich, R. .58 Ferraro, R.D. 417, 421, 440 Flower, J.W. 1105 Flynn, M.J. .697
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carter, M.B. 212, 217 Castillo, S. 408 Castro-Leon, E. 286	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. 1236 Fattouche, M. .107 Fenrich, R. .58 Ferraro, R.D. .417, 421, 440 Flower, J.W. .1105 Flynn, M.J. .697 Flynn, R.J. .1405
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carter, M.B. 212, 217 Castillo, S. 408 Castro-Leon, E. 286 Chan, M.Y. 834	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. 1236 Fattouche, M. .107 Fenrich, R. .58 Ferraro, R.D. .417, 421, 440 Flower, J.W. .1105 Flynn, M.J. .697 Flynn, R.J. .1405 Fortner, P. .1171
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carter, M.B. 212, 217 Castillo, S. 408 Castro-Leon, E. 286	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. .1236 Fattouche, M. .107 Fenrich, R. .58 Ferraro, R.D. .417, 421, 440 Flower, J.W. .1105 Flynn, M.J. .697 Flynn, R.J. .1405 Fortner, P. .1171 Foster, I. .1137
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carrona, E.A. 943 Carter, M.B. 212, 217 Castillo, S. 408 Castro-Leon, E. 286 Chan, M.Y. 834 Chen, S.K. 845	Ding, HQ. 389, 1295 Doorly, D.J. 101 Durham, S.D. 490 Egecioglu, Ö. 1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. .1236 Fattouche, M. .107 Fenrich, R. .58 Ferraro, R.D. .417, 421, 440 Flower, J.W. .1105 Flynn, M.J. .697 Flynn, R.J. .1405 Fortner, P. .1171 Foster, I. .1137 Fox, G.C. .78, 111, 131, 140,
Beard, R.A. 42 Belkhale, K.P. 930 Bennett, C. 530 Berryman, H. 1028, 1097 Bisiani, R. 690 Blanc, JY. 30, 777 Blume, L. 915 Bochicchio, M. 704 Bomans, L. 794 Boppana, R. 800 Bose, B. 64 Braaten, M.E. 463 Brandenburg, J.E. 1348 Burns, G.D. 790 Campbell, P.M. 943 Carter, M.B. 212, 217 Castillo, S. 408 Castro-Leon, E. 286 Chan, M.Y. 834 Chang, YC. 1465	Ding, HQ. 389, 1295 Doorly, D.J. .101 Durham, S.D. 490 Egecioglu, Ö. .1353 Eissfeller, H. .568 El-Rewini, H. .64, 1171 Elster, A.C. .311 Embrechts, H. .794, 876 Erçal, F. .364 Ewing, R.E. .530 Falsafi, B. .159 Fapojuwo, A. .107 Fatoohi, R.A. .1236 Fattouche, M. .107 Fenrich, R. .58 Ferraro, R.D. .417, 421, 440 Flower, J.W. .1105 Flynn, M.J. .697 Flynn, R.J. .1405 Fortner, P. .1171 Foster, I. .1137

Francioni, J.M	Huson, M.L
Fuchs, W.K	Hutchinson, S
Gach, M	Ikudome, K
Ganesan, S	Jackson, T.L
Gehringer, E.F	Jacobi, N
Geist, G.A	Johnsson, S.L
Geist, R.M	Jones, J.P
George, K.M	Kastner, J
Gerasoulis, A 291	Katter, Jr., O.E
Ghose, M	Kennedy, K
Gopinath, P	Khambekar, P.K
Gorrod, M.J	Khanna, A
Gottschalk, T.D	King, D
Grigg, K	King, H
Grimshaw, A	Koelbel, C
Griswold, W.G	Kolawa, A
	Kremer, U
Grit, D.H	
Grosch, C.E	Krumme, D.W
Grunwald, D.C	Kushner, E.J
Guan, X	Lamb, S
Gunter, K.G	Lamont, G.B
Gupta, A.K	Langs, D
Gupta, R	Langston, M.A
Gupta, S.N	Leaver, E.W
Gurewitz, E 78, 140, 148	Lee, DL
Gustafson, J.L	Lee, SJ
Gutt, G.M	Lee, T.C
Hadimioglu, H	Leonard, A
Haff, P.K	Leung, A.P
Hambrusch, S.E	Lewis, T.G
Hammond, W.H	Li, CC.J
Han, F	Li, P.P
Harding, W.A	Liang, C.T
Harrison, G.A	Liewer, P.C
Hartrum, T.C 1261	Lillevik, S.L
Hauptman, H	Lima, M
Hawley, L	Lin, KJ
Heath, M.T	Liu, J.WS
Hempel, R	Lockhart, T.G
Hensel, E	Lorenz, J
Hermitage, S.A	Lu, H
Hey, A.J.G	Lynch, J.D
Hinz, D.Y	Lyzenga, G.A 417
Hipes, P	Maccabe, A.B
Ho, A.W	Mack, D.A
Ho, CT	Mak, B.K
Hofman, R.F.H	Makivic, M.S
Horiike, S	Marzocca, C
Horvath, J.C 2, 117, 513	Matthews, M.M
Huisman, L	Mattisson, S
Huntsberger, B.A	Maxfield, H.A
Huntsberger, T.L 171, 206	McHenry, J.T

McKoy, V	Raja, P.V.R
McLaren, R.D	Ramanathan, J
McMillin, B 732, 968	Ramanujam, J
Mehrotra, P	Romach T 710
	Ramesh, T
Meier, D.L	Rasnow, B
Meyer, D.E	Ravishankar, M
Midkiff, S.F	Reed, D.A
Miekkala, U	Robert, Y
Miguet, S 124, 1324, 1343	Rochat, B 676
Miller, J.J	Rodriguez, J
Miller, P.R	Rogers, W.A
Miller, R	Romino C
Mirchandaney, S 1028	Romine, C
	Rommel, C.G
Mirin, A.A	Roose, D
Moon, Y	Rose, K
Morari, M	Rosing, M
Müller, S.M 340, 568	Ross, A
Nair, I	Rover, D.T
Nazief, B.A.A	Ruehli, A
Nelken, I	Ryckbosch, J.W.A
Nestlerode, W.C	Sabin, T
Neusius, C	Sadayappan, P
Nevanlinna, O	Calabara V A
Nothin D	Saletore, V.A
Notkin, D	Saltz, J 1028, 1097
O'Leary, P	Sarwal, A
O'Sullivan, M 1289	Sauermann, J
Olsson, L	Sawyer, G.A
Olster, D.B	Schmiermund, T.E 1334
Ong, E.T	Scott, D.S
Overbeek, R	Sears, M.P
Özgüner, F	Seidel, S.R
Padgett, W.J	Sen, R.K
Pargas, R.P	Sengupta, A
Parker, D.L	Shok D
Porkor IW 417 401	Shah, R
Parker, J.W	Sharp, III, H.F
Patterson, J.E	Sharpley, R.C
Paul, R.F	Shelton, W.A
Pazat, JL	Shin, K.G
Peck, J	Sigmon, K
Peercy, M	Simon, H.D
Pépin, F	Skelboe, S
Perry, L.P	Skjellum, A
Pesmajoglou, S 101	Sklansky, J
Peterson, L	Smith, II, R.J
Petherick, L	Snolisk D D 1071
Douglas D W	Snelick, R.D
Peyton, B.W	Snyder, L
Philippe, JL	Sobczak, R.S
Pinkston, T.M	Socha, D.G
Plimpton, S.J 478	Sochacki, J
Poplawski, D.A	Stallmann, M.F.M
Pramanik, P	Stevenson, D
Raghavendra, C.S	Still, C.H
	, = -=

Stocks, G.M	Wang, J
Stout, Q.F	Weaver, R.P
Strayer, W.T	Weeratunga, S
Sturtevant, J.E	Wegman, E.J
Su, W	Weide, B.W
Sykes, D.A	Welch, L.R
Taha, T.R	Westall, J
Tang, T	Wieland, F
	Wilkerson, R.W
Teague, K.A	
Thacker, W.I	Williams, R.D 470, 1246
Thomas, H	Winstead, C
Thuman, P	Womble, D.E
Toppur, R	Wong, YF
Torbjørnsen, Ø	Wright, C.T
Trystram, D	Wu, J
Trystram, D	Xu, M
Tsai, W.T	Yantchev, J.T
Tseng, P.S	Young, B.C
Tung, YW	Zemoudeh, K
Van de Velde, E.F	Zhang, H
Vandewalle, S	Zhang, X
Vaughan, C.T	Zhou, W
	Zhu, J.P
Velmurugan, D	
Vemulapati, U.B	Zipse, J.E
Villard, G	Zubair, M
Walker, D.W	



The Fifth Distributed Memory Computing Conference

1: Expert Systems

Hypercubes for Critical Space Flight Command Operations

J. C. Horvath, T. Tang, L. P. Perry, R. C. Cole

Mission Profile and Sequencing Section

D. B. Olster and J. E. Zipse

Flight Command and Data Management Systems Section Jet Propulsion Laboratory, California Institute of Technology 4800 Oak Grove Drive, Pasadena, CA 91109

Abstract

Controlling interplanetary spacecraft and planning their activities, as currently practiced, requires massive amounts of computer time and personnel. To improve this situation, it is desired to use advanced computing to speed up and automate the commanding process. Several design and prototype efforts have been underway at JPL to understand the appropriate roles for concurrent processors in future interplanetary spacecraft operations. Here we report on an effort to identify likely candidates for parallelism among existing software systems that both generate commands to be sent to the spacecraft and simulate what the spacecraft will do with these commands when it receives them. We also describe promising results from efforts to create parallel prototypes of representative portions of these software systems on the JPL/Caltech Mark III hypercube.

I. Background

Controlling interplanetary spacecraft and planning their activities, as currently practiced, requires massive amounts of computer time and personnel [1,2]. As missions become longer and more ambitious, and as budgets become tighter, it is desirable to have more autonomous spacecraft, or, at least, more automated ways of controlling spacecraft from the ground. As spacecraft become more autonomous their onboard computers become correspondingly more complex, and simulating them on the ground to predict and plan their actions becomes more difficult. It is further desired that these new software systems be "multi-mission" and able to support more than one complex spacecraft. Hence, larger ground computers and more sophisticated ground software are required to support these more advanced spacecraft. Several design and prototype efforts have been underway at JPL to understand the appropriate roles for concurrent processors in future interplanetary spacecraft operations.

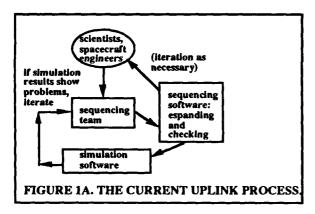
The prototypes that will be described in this paper were

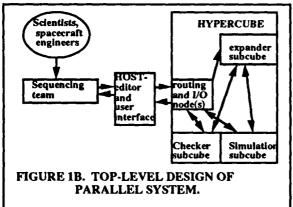
built on the JPL/Caltech Mark III hypercube[3], which is a 68020-based hypercube topology distributed memory parallel processor. The Mark III we have been using has 4 Mb per node, with no shared memory. Two 68020s are used on each node, one of which is dedicated to message routing, the other of which is used for data processing. Distributed-memory machines of topologies other than hypercubes have not been examined for this application at this time, and will not be discussed further in this paper. Other architectures are, however, being considered as well for flight versions.

Software systems have been built to support planetary missions over the years that both generate commands to be sent to deep-space spacecraft (e.g., the "SEQGEN" program) and simulate what the spacecraft will do with these commands when it receives them. These software systems have grown up around the architecture of a Unisys 1100 mainframe (including substantial implementation in assembly), and hence it will require some thought and planning to accurately port this system to distributed-memory parallel processors in a way that successfully exploits this architecture. Along with porting the software to a parallel machine, studies are being done to understand what new capabilities and system architectures are enabled by using a more capable machine, particularly a parallel one. The current mainframe sequencing environment is a heavily batch-oriented one, and a culture of spacecraft command review has grown up around the files that go into these existing programs. Hence moving to a parallel machine and building more powerful software may enable some cultural changes (and manpower savings) in the way command loads are built for spacecraft. More powerful software might also be more general and more able to be used for multiple missions, instead of largely rebuilt for each new mission.

Figure 1A shows the generic process of commanding planetary spacecraft, commonly referred to as the "uplink" process. Figure 1B shows the current top-level design of

our sequencing and simulation system, which is implemented as subcubes of a Mark III hypercube. A user employs an editor of some sophistication and consults with experts on the various spacecraft subsystems to determine a plausible set of commands. A time-ordered file is created with these commands in it. These commands are fed to the expander program. When the expander program gets the file, it expands the first time-slice of the file, and then sends the expanded output to the checker program and the simulator program. The checker program checks the output against high-level constraints ("Don't turn the star scanner within ten degrees of the sun.") The simulator program predicts what the spacecraft will actually do, at either the functional or bit level, with these commands. While the checker and simulator are processing the first time slice of commands, the second set is expanded. For future high-fidelity bit-level simulators, however, it may be necessary to expand all commands before simulating them, since memory management issues may come into play. Then it may make more sense to multitask and first use all the processors of the hypercube to perform expansion, and then use half of them for simulation and half for checking.





Simultaneous with this system-level design (the first incarnation of which was described in [4]) we are

prototyping some of the characteristic functions of these codes in parallel to see which portions benefit the most, and which would perhaps be adequate in sequential mode.

II. Parallel Command Generation

SEQGEN has several main parts. The core routines alluded to above are known as "expander" and "checker." Expander takes in high-level spacecraft "profile activities" (PAs) and outputs a time-sorted list of lower-level commands that have been "expanded" from the input PAs. A typical PA might be "take a series of pictures of Venus." A typical low-level command that might occur in that sequence might be "turn toward Venus." In flight SEQGEN, the code that tells expander which commands to generate from a given PA is written in "Sequence Component Development Language" (SCDL) . SCDL is a vaguely FORTRAN-like "little language" which is then translated into PL/1 code. It is this PL/1 code which actually runs on the Univac and generates expansions from input files of PAs. One could write the code describing how to expand PAs directly in PL/1 (or C); however since many expansion functions are used over and over, the application-tailored SCDL is more efficient for developing actual flight expansions. However, note that the translation of SCDL into the language that actually runs on the target machine is only performed rarely, and thus this translation step is not really important in our hypercube timing and feasibility design; indeed, it was avoided for the first prototype, as will be described.

The parallel version of expander described here avoided the SCDL step and had hard-wired C code that parsed various "flight-like" PAs. Each of the N nodes of the hypercube expanded to completion 1/N of the input PAs. The only inefficiencies were thus load imbalance among the nodes of the hypercube (if some PAs were more complex to expand than others) and sorting inefficiencies when the PAs expanded on different nodes were interleaved into one master file on all nodes and sorted in time order on all nodes. It became apparent that sorting was a bottleneck, and so we wrote a efficient parallel sort to decrease these inefficiencies.

This program is really a cross-compiler for the onboard computer. We have come to the conclusion that front-end compilation (without optimization) is naturally highly parallel, since each command in the input file is checked semantically and parsed without any interaction with other commands. Hence the incoming command file can be split over hypercube processors with no interaction required among the processors.

III. Integration of expander and sort

The parallel version of expander with hard-wired expansions has been integrated with a high-speed sort [5]. This sort improved the run time substantially, as shown. These are all relatively small cases so load-balancing effects dominate after four nodes. Here, the efficiency, E, is defined as:

Table 1. Times and efficiencies for expanding and sorting (times do not include I/O)

Nodes	Sequential sort		Parallel	sort
	time	E	time	E
	(s)_	(%)	(s)	(%)
1	20.94	(100%)	23.81	(88%)
2	13.30	(79%)	10.60	(99%)
4	10.02	(52%)	6.67	(78%)
8	7.58	(35%)	4.91	(53%)

The parallel sort has some overhead which slows down its one-node version from the non-parallel sort one-node version. Hence, parallel efficiencies are quoted for the parallel sort both relative to the one-node non-parallel sort case. This overhead is such that the code gives the appearance of running more than twice as fast on two nodes. This is a peculiarity of the sort implementation. From these numbers, it is apparent that a large portion of the inefficiency was indeed due to the non-parallel sort. Remaining inefficiencies are mostly of the load-balancing variety and will improve when we use large, flight-like sequences with more complex expansions than our small test demonstrations, and do not restrain ourselves to a test case that will fit on one node's worth of memory (4 Mb in the Mark III case.)

Since it is envisioned that eventually SEQGEN and a simulator might run on the same cube at the same time, the next step was to modify expander to run on just a subcube of the hypercube, instead of occupying the whole cube in single-user mode. This generalization was completed late last year. It is important that we be able to run on various parallel machines without massive rewrites; this generalization makes the code more portable.

As noted above, the translation from SCDL (in which expansions are designed) to code which can run on the host computer is not done too frequently. However, if we directly hard-wire expansions we will need to individually

code each PA expansion. Instead, we are building a simplified SCDL interpreter that produces C code that could run on either the Mark III or transputer. This SCDL interpreter will initially interpret only the subset of commands required to interpret an expansion of the Galileo profile activity which performs a spacecraft maneuver, but as time and resources is available the interpreter could grow up to full flight capability, and use the preexisting PA definitions that were built in SCDL for the Unisys. This will let us know if we are scalable to full flight capability.

Since all expansion of given PAs is done fully independently and in parallel, and since all nodes have knowledge of all the input PAs (although not their expansions) it is not necessary to add any parallel features to the expander C code generated by the SCDL interpreter and hence this portion of the code will port nearly automatically to transputer or Mark III hypercube. The load balancing and sorting portions of the code will need to be adapted slightly, however, should we port the current Mark III expander to a transputer, or other machine. Note that distributed memory is not a disadvantage with this application and may indeed be an advantage in this portion of the code.

IV. Command Checking in Parallel

A design was developed for a parallel implementation of SEQGEN checker, which takes the low-level commands generated by the SEQGEN expander software described above and checks them against a matrix of constraints. A design study was performed to understand the best mapping of this problem to the hypercube, and both Time Warp and Chandy-Misra implementations were considered. A recommendation was made to use Chandy-Misra algorithms for a future prototype, since this paradigm lends itself better to the network-style interactions found in Checker.

Checker takes the list of low-level commands generated by expander and checks the commands against a matrix of constraints. The constraints are also specified by a flight project in another "little language", this time a vaguely LISP-like one. This parser changes constraints written in this language into code that can be used by the host computer (PL/1 in the case of the Univac flight version). Hence, the translation into C code can be done once and the checking network will remain the same for long periods of time. This will allow conventional optimization techniques to be used to map the constraint network to the hypercube nodes in the most efficient possible way.

IV.A. Sequential Discrete Event Simulation Checker can be thought of as a discrete event simulator; it takes each command to be executed at a discrete point in time and models the spacecraft functions driven by that command. All discrete event simulators operate on sequentially ordered event lists [6]. In this case, the command sequence is the event list and the command execution time is the simulation time. The simulator takes the event of lowest simulation time and removes it from the list. It performs all computations defined for that event and effects changes on the system being simulated (here, a spacecraft). This event may generate other events for future simulation which are placed on the event list in time order. The simulator then moves to the next event and updates the simulation time.

Since the internal spacecraft structure consists of electronic circuits, Checker functions to a large extent as a logic circuit analyzer [7]. A circuit node is defined using a constraint network, the set of functions that simulates the state of the spacecraft. The command is physically an electronic pulse and its state is set to true. This triggers the constraint network functions that update the circuit's state, and verify rules and constraints. This information may be used as input for all circuit components (or nodes) that the command affects. Then the state of that command is set to false and the process repeated until all nodes have been analyzed. Some nodes may call models which simulate spacecraft operations. Models take the form of commands local to SEQGEN and are placed on a time-ordered event list. These commands are checked in the same manner as sequence commands. necessary, constraint network writes error or status messages to the SEQGEN's output files.

Checker's similarity to logic circuit simulators makes it especially adaptable to a parallel implementation. Constraints in separate sections of the circuit need not be checked sequentially since they do not affect each other immediately (although they may interconnect elsewhere). Parallel logic circuit simulation has already been implemented successfully on several multiprocessors, such as the Intel iPSC and Ametek Series 2010, and should demonstrate similar speedups on the hypercube [8].

IV.B. Parallel Discrete Event Simulation Paradigms

One way to parallelize SEQGEN checker would be to divide sequential procedures onto separate processors. However, finding the inherent parallelism in simulation code is difficult and usually inefficient. In this case, it is not at all feasible since the JPL hypercube does not support the PL/1 or Univac assembly code in which SEQGEN was originally written. The Chandy-Misra and

Time Warp algorithms, on the other hand, take advantage of the parallelism inherent in the system being simulated. Logic circuits, for instance, can be partitioned very easily even to the point where each processor simulates one circuit node.

In parallel discrete event simulation, the repeated manipulation of a sequential event list would inhibit concurrency [9]. Therefore, events are divided among the processors. If one event affects another it sends a timestamped message along the communication links. The processor receiving this message can accept it immediately or place it in an input queue. parallel simulation to be consistent with its sequential counterpart, the system must be deterministic which means that the current state of the system uniquely determines the next state. Otherwise event messages may not cause the correct event to be simulated, or the simulation halts when a processor is unable to make a decision. The simulation proceeds by synchronizing events. Time-Warp and Chandy-Misra have different synchronization schemes which allow speedups only when their particular applications meets certain criteria. Which algorithm works better is usually application dependent.

IV.B.1. Time Warp

The Time Warp mechanism is operational on the JPL/Caltech Mark III hypercube and was therefore seriously considered for implementing spacecraft constraint checking. In Time Warp, events are distributed among the processors and communicate with timestamped messages [10]. Each processor stores messages in an input queue sorted by time. The message with the smallest timestamp is simulated and the Local Virtual Time (LVT) is advanced to that timestamp. At any point, one processor may have an LVT ahead of the others, but this does not affect the simulation except to determine the timestamp of a message. A process does not concern itself with the possibility of receiving future events; this is considered an optimistic approach. The state of each event is saved in case a processor should receive an event with a timestamp less than the LVT. If it does, the simulation rolls back to the event just previous to that timestamp and resimulates. For example, if a message with the timestamp 9.5 is simulated and then the message with timestamp 8.2 reaches the input queue next, a rollback will occur.

After rollback, messages that had been previously sent with timestamps later than the rollback timestamp are obsolete and must be removed. The processor sends an anti-message which contains the same information as the original message, but will cause both to be removed from an input queue. If the original message has already been

simulated, or hasn't arrived yet, the anti-message is processed. Upon discovering a duplicate event in the state queue, an event will trigger another rollback which then, in turn, may cause more rollbacks. How then, does the simulation progress? Each processor also keeps track of the Global Virtual Time (GVT) which is the lowest timestamp of every message or event in the system. No processor can rollback prior to the GVT, and all event states with timestamps less than the GVT can be removed. Therefore, Time Warp is guaranteed to progress. Time Warp terminates only when all events have been simulated and all messages and anti-messages processed.

IV.B.2. Chandy-Misra

The Chandy-Misra simulation proceeds conservatively. A processor does not begin computation until it is sure that it will not receive any messages with a timestamp less that its local simulation time [9]. Therefore, the simulation does not require a global time clock or more memory than necessary for the sequential simulation. Events, in this case called logical processes (LP), are distributed among the processors. The LP interdependencies are mapped so that a processor will send and accept messages from certain processors with as many input and output queues. The LP waits until event messages are received from all input links. In this way, it ensures that no future messages will be received with timestamps less than those of the inputs. The event of minimum timestamp is simulated and the local simulation time updated to this timestamp. Then the LP will wait to output a message until the receiving LP expects it. All future messages will have a greater timestamp.

The simulation may deadlock if a processor waits to send or receive messages but is blocked. Deadlock can be prevented using null messages which serve to update the simulation time and free waiting processes. However, if the computation time of an event exceeds the time for another process to produce a null message, the system may become overloaded with null messages. Though this cannot be avoided if there are too many long computations, some null messages may be delayed and bundled until they can be send with the next event message. This still prevents deadlock and reduces the number of messages. Deadlock can also be allowed to occur and then overcome using an algorithm which determines the cause of deadlock and how to recover from it. This scheme has a computation time overhead which may be acceptable depending on how often the system deadlocks. The simulation terminates when it has processed all events and messages.

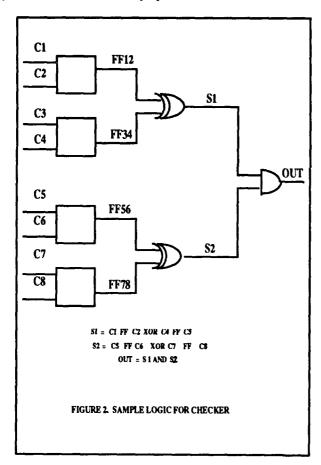
IV.C. Checker Implementation

The first and most difficult consideration of writing Checker in parallel is determining how to distribute the circuit nodes among the processors. Load-balancing requires an analysis of the circuit nodes, their associated constraints and the event list. Either all of the circuit or only the sections being checked can be partitioned and sent to the processors.

Figure 2 shows an example of Checker logic. The entire circuit may be placed on one processor, or the top and bottom halves on two separate processors, and node "OUT" on a third, etc. What processor a circuit node would be sent to depends on how many constraints it has, how long it takes to check those constraints, how many times it is evaluated, and what other nodes it affects (to avoid interprocessor communication). information would be determined ahead of time, and circuit nodes would be statically allocated to a processor. (The current flight SEQGEN checker has a constraint network that does not change from run to run, and hence this mapping could be done very optimally, with a simulated annealing or other approach.) Later it might be possible to implement dynamic load-balancing so that an idle processor could take on another's work. However, this might incur a huge overhead in information transfer. Another large overhead would be repeated file manipulation. In fact, earlier versions of Time Warp did not support file handling because rollbacks require messages to be buffered and written sequentially in global time order. Checker writes each command to SEQGEN's output files; it may also write status and error messages, but only if certain conditions are met. How often a processor writes to a file greatly affects its load. Once the synchronization scheme has been implemented, we will investigate various load-balancing strategies to optimize the simulation.

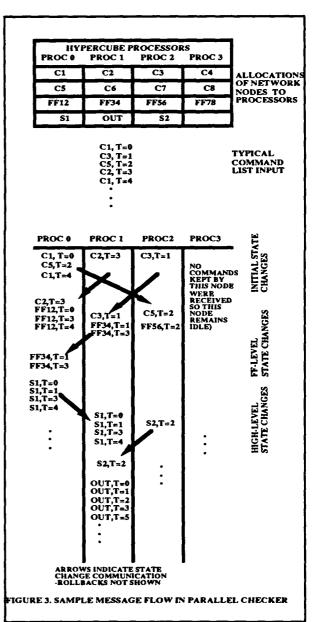
Choosing between the Time Warp and Chandy-Misra algorithms proved to be difficult since the constraint network is large and complex with many different types of computations. Knowing exactly how the system behaves is essential in determining which synchronization scheme is appropriate. Figure 3 shows the structure of constraint checking using Time Warp. The circuit being checked is that of Figure 2, where all circuit nodes are completely distributed among the four processors. (Note the time tags on the commands.) Individual commands to be checked are assigned to one of the processors in this case. and instances of changes in state of these network nodes needs to be transmitted across the hypercube processors, as shown with the heavy arrows in Figure 3. Note that although there is an even distribution of network nodes across the hypercube processors, Node 3 remains idle since no commands that it handles were in the input command file. This is an inherent risk with this parallelization; however the other possibility (having the entire network on every node with commands randomly assigned to processors, as is done in expander) is unwieldy to implement owing to the high interconnection of the network.

At first it would seem that saving all states would take up too much memory, but that is solved by removing all states before the GVT. This particular example shows the bottleneck that can occur by simulating only a small sequence of commands. However, the average number of commands in a sequence ranges from 1000 to 5000 and this bottleneck would not be as likely to occur as long as the command assignments of command types to processors were reasonably optimal.



Although Time Warp could easily support spacecraft constraint checking, several factors make it unsuitable. Time Warp is a general simulation operating system that would have to be adapted to the specific requirements and intricacies of constraint checking. It produces significant speedups only when rollbacks do not present too much

overhead. This is usually true for simulation of physical systems where a guess in behavior does not cause too many rollbacks. Physical systems involving large computations allow most messages affecting an event to arrive before an incorrect choice can be made. However, static logic circuit verification involves little computation per node. It does take time to verify complicated circuits that may have thousands of nodes which is what makes a parallel simulation worthwhile. If constraint checking added a large amount of extra computation time, the simulation would work well with Time Warp. We tested several different constraints on the hypercube and they took very little time to process.



When simulating the same system as shown in Figures 2 and 3 for Time Warp, if there are only a few commands, the system deadlocks very quickly if null messages are not sent or a detection and recovery algorithm used. Otherwise the logical process FF12 has to wait until it receives both C1 at time 0 and C2 at time 3. The Chandy-Misra method has already been adapted for logic circuit simulation, though not strictly to test circuits, but to test the algorithm itself. Therefore, the major drawback in using the Chandy-Misra method is that no implementation exists on the JPL/Caltech Mark III However, in writing it from scratch, eventually the simulation would be especially efficient for spacecraft constraint checking. But if the circuit is so interconnected with multiple outputs and feedback that it reduces the inherent parallelism of the system, then Chandy-Misra would essentially lock-step through the simulation and Time Warp would be the only alternative. Evidence indicates that this is not the case, and a simplified subset of Chandy-Misra will be implemented.

The prototype will simulate a small subset of commands -- the same set being used as an example by expander, allowing the integration of the two protetypes. Constraint network definitions will be statically distributed to the processors and each command from the sequence will be sent to the processor containing its definition. Then the simulation will proceed according to the Chandy-Misra algorithm. Though deadlock avoidance using null messages is simpler to program than the deadlock avoidance and recovery algorithm, we will investigate which scheme is more efficient. Later more constraints and commands will be added. Previous results with parallel logic circuit simulations have shown that the simulation will be most efficient when a large set of commands is used to reduce the number of idle processors [8]. The prototype implementation of Chandy-Misra will serve to test constraint network behavior, since it will demonstrate significant speedups only when a large portion of constraint network has been adapted for the hypercube.

How efficient the parallel implementation of Checker proves to be is dependent on the sequence and its associated models. Checker might demonstrate significant speedups for some spacecraft models such as simple light-switch relay circuits, but may even slow down the checking time for a complicated, highly-interconnected constraint network such as the one modeling the restrictions on a spacecraft maneuver. But since the spacecraft is comprised largely of simple models, it would be possible to run Checker in parallel for most sequences and sequentially for the few remaining sequences and still work efficiently.

One of the major portions of flight SEQGEN checker is a "little language" parser, which translate the LISP-like constraint network specifications into the C which runs on the hypercube nodes. A subset of this parser will be needed for a checker prototype that can perform the constraint checks for Profile Activities that are being used as tests for the expander module. Building this realistic prototype will prove that the checker can take the results of the expander and check them in parallel.

V. Spacecraft Simulation in Parallel

An existing VAX high-level simulation of some of the functions of the Galileo spacecraft onboard computer was ported to the hypercube. This particular simulation turned out to run very quickly on one node, making parallelization unnecessary. The reasons why this was true give insight into design requirements for future simulations. The Galileo onboard computer is itself a parallel computer (although of the master/slave variety), and the VAX simulation took this parallel computer and simulated it sequentially. A large portion of the code was bookkeeping to accomplish this simulation of a parallel computer. Therefore, if such a simulation is desired, it should not be ported from a sequential simulation of a parallel system but should be written in parallel in the first place. Some observations along these lines will be discussed. Plans are in place to boild a simulation of the full capabilities of the Galileo onboard computers.

VI. Flight-qualifying parallel code

An attempt was made to understand what it will take in the way of new software testing methods and design tools to certify hypercube code (and hypercube operating systems) for flight-critical systems to the same standards that are now applied to sequential codes. Parallel codes have several unique characteristics that make them challenging to debug and to test fully. In particular, although any one module may be tested thoroughly, exactly when and how and with which other program running on which other node it will interact is difficult to predict. Quasi-parallel onboard computers, like the Galileo and Magellan onboard computers, get around these difficulties with architectures that are both synchronous, are not interrupt-driven, and are controlled by one master processor (although which processor is the master can change over time.)

In parallel code, three major classes of software failures can be encountered: the hard failure (where the software hangs), the soft failure (where the software continues to process, but gives the wrong numerical result), and the algorithmic failure (a "bug" of the same sort as one

encounters in sequential coding.) Typically, one will build a sequential version of the code which removes most of the latter bugs, but some new ones inevitably show up as interaction phenomena when the software is run in parallel for the first time. It has been the first author's experience that most interaction bugs show up on two processors and can be removed there, and that virtually all show up on four processors. However, it has occasionally been true that certain bugs show up on eight or sixteen processor or larger cubes only, and these bugs are frequently subtle and intermittent in their symptoms.

By their nature, synchronous codes running under the Crystalline Operating System (CrOS) or its commercial variations have a priori predictable communication patterns, since interrupt-driven communications are not allowed and processors are required to handshake in a predictable pattern, or deadlock results. Synchronous programs, therefore, are more prone to the hard failure during the development phase while these communication patterns are being debugged. Hard failures do have the virtue that they are usually easier to spot than the soft failure; however, they are more difficult to debug since the cube will simply stop in most cases with no diagnostic output. Sometimes these failures will not show up until perverse data is processed by the code. The general synchronous sorting routine described above, for example, initially failed when presented with unbalanced partial lists to sort on different nodes; the algorithm had assumed balanced lists to sort across the cube. This was corrected, but at the cost of more complexity, which in turn exposes the code to failures of the other two types.

Purely asynchronous code is probably the most difficult for the design of software qualification tests and criteria for its acceptance, since the possible interactions of the software with itself and with the commands it is checking are so numerous. This will particularly be a problem for checking algorithms, which tend to naturally be somewhat asynchronous and hard to design even sequentially (since more sophisticated ones verge on expert systems.)

We will need to design cases that test communication paradigms, and not just output. This is analogous to the process in expert system debugging where both the reasoning that produced a certain result as well as the result itself both need to be correct before the system can be accepted for critical applications [11]. For example, the commonly-used flight software testing technique of regression testing would need to be expanded to test communications as well. In a sequential code, testing four sets of the same command to be expanded might not be interesting, but might be valuable in parallel to test whether all processors take care of the expansion the same

way.

VII. Open issues and conclusions

Many open issues remain in the use of hypercubes for critical operations. Issues of reliability and predictability need to be understood more thoroughly. However we feel that parallel computers in general and hypercubes have a role in spacecraft commanding and other critical applications, but many challenges remain to make these codes both powerful and reliable. Although this study is geared to the particular problem of generating commands for, and predicting the actions of, a deep-space probe, the conclusions are applicable to efforts to generate and verify code for any critical computer, remote or not.

VIII. Acknowledgements

Many people have contributed to the work described in this paper; space does not permit naming them all. In particular, the authors would like to acknowledge the JPL Hypercube Project and Dave Curkendall for making hypercube time available for the prototypes. Nooshin Meshkaty and Edith Huang gave invaluable user support. Barbara Zimmerman, John Flower, gave us assistance on various portions of this task; Phil Hontalas, Matthew Presley and Ed Upchurch on Time Warp; and Wen-King Su and Mani Chandy on Chandy-Misra. L. Perry's research was conducted through the Caltech Summer Undergraduate Research Fellowship. The work described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

IX. References

- [1] Linick, T.D., "Spacecraft Commanding for Unmanned Planetary Missions: The Uplink Process", Journal of the British Interplanetary Society, Vol. 28, No. 10, 1985.
- [2] McLaughlin, W.I., and Wolff, D.M., "Automating the Uplink Process for Planetary Missions." AIAA-89-0580, 27th Aerospace Sci Mtg, Jan 9-12 1989, Reno NV.
- [3] Burns, P., Crichton, J., Curkendall, D., Eng, B., Goodhart, C., Lee, R., Livingston, R., Peterson, J., Pniel, M., Tuazon, J., and Zimmerman, B., "The JPL/Caltech Mark III fp Hypercube." Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Pasadena CA Jan.

1988, p.872.

- [4] Horvath, J.C. and Cole, R.C., "Spacecraft Sequencing on the Hypercube Concurrent Processor", Fourth Conference on Hypercubes, Concurrent Computers and Applications, Monterey, CA, Mar. 1989.
- [5] Tang, T. "Parallel Sorting on the Hypercube Concurrent Processor." Presented at DMCC5, Charleston, SC, April 1990.
- [6] Samadi, B., "Distributed Simulation, Algorithms and Performance Analysis", Ph.D. Thesis, UCLA, 1985.
- [7] Cole, R.C. and Crichton, G.A., "Galileo User's Guide: SEQGEN Program Set", JPL internal document D-263, May 1985.
- [8] Su, W. and Seitz, C.L., "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm", Caltech CS Technical Report TR-88-22, Dec. 1988.
- [9] Chandy, K.M. and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Communications of the ACM, Vol. 24, No. 11, April 1981.
- [10] Jefferson, D., et.al, "The Status of the Time Warp Operating System", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Pasadena CA Jan. 1988, p.738.
- [11] Stagler, R., "The Problem of Certification for Expert Systems." JPL internal document, October 1988.

A Massively Parallel Expert System Architecture for Chemical Structure Analysis

Ronald S. Sobczak
Manton M. Matthews
Department of Computer Science
University of South Carolina
Columbia, SC 29208

1 Introduction

This paper discusses a novel approach to solving properly asynchronous heterogeneous problems on a hypercube architecture such as the NCUBE/10 computer. The discussion is divided into the classification of possible problems on the hypercube, a description of blackboards and their utility in solving properly asynchronous heterogeneous problems on the NCUBE, the application of these techniques to a specific example, structure elucidation of organic compounds from spectroscopic data.

The NCUBE/10 [13, 15] is a MIMD computer consisting of 1024 32-bit processors and a coarse grained homogeneous distributed memory. The types of problems can be divided and described in terms of how the nodes communicate and how the problem is subdivided between the individual nodes [6]. Communication can be synchronous, loosely synchronous or asynchronous. In synchronous communication, all processors are doing the same thing at the same time and as a result, all communication is automatically synchronized. In properly loosely synchronous problems, each processor is doing its own thing but must synchronize with all other processors whenever doing interprocessor communication. In properly asynchronous problems, each processor acts independently, communicating whenever necessary without regard for synchronizing with the other processors. In a sequential asynchronous problem, there is a lockstep order which allows no parallelization of the problem whereas concurrent asynchronous problems can be processed in parallel. Amdahl's Law

states that the maximum speedup for any problem is always bounded by 1/s where s is the time for the serial work fraction (that portion which cannot be parallelized). Asynchronous problems are the most difficult problems to execute efficiently in an MIMD environment and therefore offer the greatest challenge to the programmer. According to Fox, out of 84 programs reviewed only 8 properly asynchronous problems were identified. Properly asynchronous problems contain the greatest uncertainty for speedup on a machine like the Ncube. No matter what type of problem, it is essential to minimize communication and when it is necessary, to restrict it to adjacent processing nodes with relatively large program segments working largely independently if at all possible.

Synchronous communication implies that all processors are executing identical code on different segments of the data. Asynchronous communication may also involve identical code but because of differences in the complexity of the data, the processors may not complete their assignments at the same time. On the other hand, it is feasible that different processors may actually be executing different code simultaneously. Fox has "ruled out course grained functional decomposition, e.g., different sub-routines of a given application running on different nodes, because this is only capable of obtaining modest speedup as essentially all real applications only have a few distinct actions to be performed currently." We are interested in just how much speedup is possible in a problem which involves asynchronous communication between different segments of code executing simultaneously.

2 Blackboard Systems

Our approach for solving heterogeneous problems with properly asynchronous communication on a parallel computer is to use a blackboard model for control of the different code executing simultaneously and for communication of the partial solutions between the different code segments. A blackboard [22] is a problem-solving model that allows opportunistic reasoning. The blackboard model consists of three major components: control, the blackboard data structure and separate knowledge sources.

Control dynamically selects which knowledge sources to execute and which data to manipulate at any one time, thereby coordinating the manipulation of the blackboard and determining the focus of attention. This is especially useful for complex ill-structured problems with poorly defined goals and an absence of a predetermined decision path which is a good description of how a heterogeneous asynchronous problem must be solved in a parallel environment. Because control is dynamic, it can utilize opportunistic reasoning techniques and avoid lock-step control sequences which would make execution inefficient. Alternate solutions can develop simultaneously and heuristic methods can be used to short-circuit computation. Forward and backward reasoning can be used simultaneously. Control maintains criteria for determining when to terminate execution and is capable of handling errors gracefully.

The blackboard data structure is a global database. The individual knowledge sources communicate and interact via the blackboard. The solution space includes all possible partial and full solutions. It can be organized into one or more application-dependent hierarchies. Each level of the hierarchy can contain its own unique vocabulary. Ultimately these vocabularies will coalesce as the solutions develop and progress up the hierarchy. Reasoning which supports the partial solutions, can come from below and/or above in the hierarchy. The blackboard handles all message-passing constraints and allows communication be-

tween disparate sources of information regardless of the different vocabularies involved. Ultimately this should minimize communication between specialists so that the individual processors can concentrate on computation.

In order to limit interaction between knowledge sources, the problem is decomposed into loosely coupled subproblems. The individual knowledge sources can be implemented as rules, objects or procedures. Each knowledge source knows what it is capable of contributing to the solution. Details of the task dictate the type of knowledge representation and the reasoning methods employed. The interaction is organized hierarchically, with integration of diverse concepts and vocabulary. Each knowledge source can be modified without affecting the other sources, making it relatively easy to update the knowledge base. Since each knowledge source works relatively independently no one piece of data becomes a barrier to the solution but additional information will improve performance. Any uncertainty is handled with credibility weights. Conflicting data can either be eliminated if the difference in certainty is large or both partial solutions can be saved independently on the blackboard for further processing.

Blackboards are especially effective at handling problems with large solution space, dependent on noisy and unreliable data. A variety of input data can be handled. Multiple reasoning methods can be used simultaneously and solutions can develop in stages. The blackboard is a potentially effective method for finding and expressing parallelism in a heterogeneous asynchronous problem. The knowledge sources can be divided to provide optimal grain size of data and knowledge. The knowledge sources know what they can do and can be responsible for local control while Control is responsible for global control.

3 The Structure Analysis Problem

We have chosen a problem which should be an effective test of using a blackboard to maximize concurrency for a heterogeneous asynchronous problem. Organic chemists are always concerned with

determining the chemical structure of unknown organic compounds. In the past this was done by using a battery of chemical tests but these tests are messy, time consuming and destroy the sample in the process. Modern organic structure elucidation depends heavily on absorption spectroscopy. In absorption spectroscopy, a specific frequency of light is passed through the organic compound to determine whether the light is absorbed or not. Measuring the absorption gives a variety of information about the compound. The advantages of these techniques is that they are quick, relatively easy and do not destroy the sample. The disadvantage is that the resulting data must be analyzed by an "expert" We intend to develop an expert system which can run effectively on the hypercube. Some of the characteristics of this problem are as follows. There are several different types of spectroscopy which look very different, and give very different information. In other words, they involve very different expertise and very different "vocabularies" which must somehow be integrated to generate the structure. Each spectrum may include many different absorptions. Much of the information in the spectra is uncertain and ambiguous. Therefore a human expert will process the most important and most reliable information in each spectrum first. If a structure can be determined, no further processing is necessary. On the other hand, if the solution is incomplete, further processing of the data can be done. Some of the ambiguity in the data is alleviated by the fact that data from different spectra can reinforce each other. If data from different spectra conflict, the data with the greater reliability are used first. If this fails, the data can be reevaluated. A chemist may not have all the data that would be useful. Each facility will have different equipment with limits on their capabilities. Therefore a human expert must be flexible in what data and what order is used to solve the problem.

To handle these problems, the computer expert must be modular so that individual experts can work independently on the different spectra. This also makes it easier to update and add new experts without affecting the performance of existing experts. Computer programs are often designed in a very sequential manner where each set of data is exhausted before moving on to other sets. As already indicated, this is counterproductive with spectra. The blackboard approach will allow a much more flexible manipulation of the spectra, allowing for opportunistic reasoning similar to a human expert's approach. Since the blackboard allows several different solution paths to develop simultaneously, no one piece of spectral data will inhibit the progress of the program.

4 The Knowledge Sources

The following is a description of some of the experts involved in the elucidation of chemical structure. The knowledge sources can be divided into three groups of experts, the structure generation routines, spectral experts, and chemical experts.

Structures are generated in stages [8]. different stages can be executing simultaneously since more than one structure is often possible. These routines are modeled after the design of CHEMICS. The following are a list of primary components which are basic components for constructing organic molecules: CH₃, CH₂, CH, C, CO, OH, O, NH2, NH, N, SH, S, F, Cl, Br, and I. Secondary components are combinations of primary components useful in constructing organic molecules and that can be related to spectral data. There are 86 secondary components. Tertiary components consist of a secondary component combined with an "afferent nature" which is simply a restriction on what the secondary component can bond to. There are 630 such combinations. Initially maximum and minimum values for primary and secondary components are determined from the molecular formula. All possible sets of primary components are determined. Based on these, the possible sets of secondary components are generated. The secondary components are reviewed for consistency with the chemical formula and spectral data. Finally tertiary component sets are generated. The sets of tertiary components are used to generate complete structures including the bonding of the components. There are a variety of steps done including generation of linkage, tests for absolute linkage and absolute nonlinkage, checking of separated structures and checking of generated structures.

Common spectroscopy techniques include infrared, ultraviolet, proton NMR, carbon-13 NMR, and mass spectroscopy. The characteristics of an infrared spectrum include frequency in reciprocal centimeters, the intensity (strong, medium, weak) and shape (broad) of the peaks. The infrared is particularly important in determining the presence or absence of specific functional groups such as carbonyl (C=O), hydroxyl (OH), amino (NH), nitrile (CN), and carbon-carbon double and triple bonds.

Mass spectroscopy gives entirely different results. It is a bar graph where the one coordinate corresponds to mass to charge ratio and the second relates to the abundance of the ion. Mass spectroscopy also contains a large number of peaks where many of the peaks are often ignored in determining chemical structure. The mass of the molecular ion is the molecular weight of the molecule which makes it particularly important.

Ultraviolet spectroscopy is much simpler but also much less useful than either infrared or mass spectroscopy. An absorption in the ultraviolet indicates conjugation (alternating double and single bonds). Usually there are only a few absorptions or possibly none. Analysis depends on the frequency and the intensity of the peak.

Proton NMR (nuclear magnetic resonance) spectroscopy can be analyzed based on the chemical shift in parts per million or ppm (chemical environment of the different hydrogens in the molecule), integration (ratio of different hydrogens in the molecule), and splitting pattern (number of adjacent hydrogens). It contains large amount of useful information and all absorptions are significant in structure determination.

Carbon-13 NMR is a source of information about the carbons in the molecule just as proton NMR is a source of information about the hydrogens in the molecule. Chemical shift and splitting are available but integration is not. The range of chemical shift is over 200 ppm (unlike proton NMR where overlap often occurs). As a result there is very little chance for overlap between chemically different carbons.

Other spectroscopy techniques are continually being developed and refined. Our system will al-

low easy modification and addition to the experts without disrupting the system. Each spectroscopy expert must handle very different data as indicated above but the analysis of each results in molecular fragments. Therefore the blackboard will allow the individual experts to use their own unique vocabulary and ultimately convert it into a universal vocabulary of molecular fragments which are then used to direct and restrict the structure generation routines. As the experts generate components, these can be used to prune primary, secondary and tertiary components thereby preventing combinatorial explosion. If fragments from different experts conflict, the fragments with the highest certainty factor will be added to the active fragment list and the other fragments will be retained on an inactive list. The inactive list will be used only if the active list fails in generating a structure(s) which adequately explains the data. Control will use the spectroscopy experts not only to generate possible fragments but also to test generated structures to see that they are consistent with the spectral data. Therefore the spectroscopy expert will be used both at the front end and at the back end and will be an integral part in determining when to terminate execution. Like any good expert, the system will also be able to determine when there is insufficient information to identify one structure as the correct one.

An example of the solution of a spectroscopy problem is as follows: (This problem is included to indicate the variety of information which must be integrated in a structure elucidation problem. The actual details would probably be comprehensible and interesting only to another chemist.) The molecular formula for the unknown is $C_7H_{12}O_4$. The primary structure generator determined that there were 93 possible combinations or sets of primary structures. Using these as the starting point, 497 sets of secondary structures were produced as possible candidates using the secondary structure generator. All possible combinations of these would then be tested in determining the tertiary structures and the complete structural formulas that were possible. Alternatively the spectroscopy experts could be used to prune the primary sets, drastically reducing the number of possibilities.

The following information was obtained from

spectra. The Infrared spectrum for this unknown contained at least 15 distinct peaks (or absorptions) but two were particularly important, a broad absorption at 3000 cm-1 and a strong peak at 1725 cm-1. This was indicative of the -OH and C=O of a carboxylic acid group. The mass spectrum contained approximately 30 peaks with the parent or molecular ion being at 160 m/e. This could be used to determine the molecular formula which was $C_7H_{12}O_4$. The ultraviolet spectrum contained no significant absorption which indicated that the molecule lacked conjugation (alternating carbon-carbon double and single bonds). The carbon-13 NMR contained 7 absorptions. The first two came at about 180 ppm offset from TMS which suggests two different carbonyl carbons (C=O). The next peak at 70 ppm was due to the solvent used (dioxan). The last four peaks were in order a singlet (C), triplet (CH_2) , triplet (CH_2) and quartet (CH_3) in the off-resonance decoupled spectrum. Finally the proton NMR contained a peak for HOD (indicating an exchangeable proton in the molecule), a singlet for the solvent (dioxan), two multiplets integrating for two protons each (CH_2-CH_2) and a singlet integrating for six protons (two methyl groups).

Using all this information to prune the sets of primary structures leaves only one possibility, the set containing the following primary components: 1 methyl (CH_3) , 2 carbonyl groups, 2 hydroxyl groups, (combining these give you two carboxylic acid groups, which are examples of secondary structures), 1 carbon (C), and 2 methylenes (CH_2) . When combined, the only feasible structure is $(CH_3)2C(CO2H)CH_2CH_2CO2H$. This dramatically reduces the number of possibilities that must be explored.

In some problems, the available information will be less restrictive. As a result, the structure generator will have much more work to do. In these cases, it is feasible for Control to divide the work of the structure generator between several processing nodes. The following example demonstrates this:

The molecular formula for the unknown is $C_{12}H_{18}O_2$. The primary structure generator produced hundreds of primary components that were consistent with this formula. Without any restrictions generated by the spectroscopy experts, over

100,000 sets of secondary components were constructed. The spectral data did give the following information: In the infrared, an absorption at 1750 cm-1 indicated that the molecule contained a carbonyl (C=O). The mass spectrum had a molecular ion of 196 consistent with the above molecular formula. There was no absorption in the ultraviolet spectrum indicating a lack of conjugation. In the proton NMR, there were four vinyl protons, two of which were split into a doublet (indicating a hydrogen on the adjacent carbon atom). The rest of the protons all appeared at roughly the same location allowing for little additional information. Therefore the major restriction put on the structure generator was that the possible sets of primary components must contain at least one carbonyl.

Utilizing a blackboard based architecture for expert systems allows the parallelization of heterogeneous asynchronous problems. A set of nodes is dedicated to updating the blackboard, overseeing communication between the nodes and the blackboard and controlling what the individual nodes are executing at any one moment. The program is implemented on a cube of size 2n, which is subdivided into two cubes of size n. The first is the blackboard cube or bcube which, as the control unit, is responsible for maintaining and updating the blackboard, controlling how the problem is subdivided, determining the focus of attention in each module, handling any dynamic load balancing as the solution progresses and acting as the communication link between the different heterogeneous segments of the problem. The second cube of size n consists of the processing nodes each of which is directly connected to one of the nodes in the bcube. The processing nodes act collectively as the knowledge sources. All communication is transparent to the processing nodes except for their contact with the blackboard node directly connected to them.

In the following table we summarize the performance of the system for the molecular formula $C_7H_{12}O_4$. In this case there were 93 primary component vectors and 497 secondary vectors. The data in the table does not take into consideration communication time of results back to the host.

Expert Processors	Time	Speedup
1	450	
4	115	3.9
8	60	7.5
16	38	11.0
32	26	17.3
64	25	18.0

Figure 1: Times for $C_7H_{12}O_4$

5 Conclusions

In summary, successful structure elucidation requires the combination of several different expertises such as the chemical knowledge of how molecules are constructed from simple components and how this process can be restricted based on different spectroscopic data. The different structure generation routines can run simultaneously since many different pathes must be pursued. To prevent combinatorial explosion, the spectroscopy experts can simultaneously determine restrictions on the structure generation routines and check the resulting partial and full solutions. Control can dynamically reallocate processors to different subproblems. Parts of the structure generation routines can be further subdivided into homogeneous subproblems. Although a problem such as this will never attain the efficiency of a simultaneous homogeneous problem where all processors are executing the same code on different data, our goal is not to compete with such problems but to show that parallel processing on a hypercube can dramatically speed up the execution of problems which were originally considered inappropriate for such an architecture.

References

- [1] NCUBE Users Handbook.
- [2] "Parallel Processing with Large-Grain Data Flow techniques", R.G. Babb II, IEEE Computer July 1984, p55-61.

- [3] "Multidimensional Spectroscopy", W. Bremser, W. Fachinger, Magnetic Resonance in Chemistry, Vol. 23, #12, 1056-1071, 1985.
- [4] "GENOA: A Computer Program for Structure Elucidation Utilizing Overlapping and Alternative Substructures", R.E. Carhart, D.H. Smith, N.A.B. Gray, J.G. Nourse, C. Djerassi, J. Org. Chem. 1981, 46, 1708-1718.
- [5] "Structure Generation by reduction: A New Strategy for Computer Assisted Structure Elucidation", B.D. Christie, M.E. Munk, J. Chem. Inf. Comput. Sci. 1988, 28, 87-93.
- [6] "What Have We Learnt from Using Real Parallel Machines to Solve Real Problems?", G.C.Fox, 1988, p897-955, Third Conference on Hypercube Concurrent Computers and Applications.
- [7] "Further Development of Structure Generation in the Automated Structure Elucidation System CHEMICS" K. Funatsu, N. Miyabayashi, S. Sasaki, J. Chem. Inf. Comut. Sci. 1988, 28, 18-28.
- [8] "Introduction of two-Dimensional NMR Spectral Information to an Automated Structure Elucidation System, CHEMICS. Utilization of 2D-Inadequate Information", K. Funatsu, Y. Susuta, S. Sasaki, J. Chem. Inf. Comput. Sci. 1989, 29, 6-11.
- [9] "Large-Scale Concurrent Computing in Artificial Intelligence research", L. Gasser, p1342-1351.
- [10] "Parallel Algorithms and Architectures for Rule-Based Systems", A. Gupta, C. Forgy, A. Newell, R. Wedig, p28-37.
- [11] "Development of Parallel Methods for 1024-Processor Hypercube", J.L. Gustofsen, G.R. Montry, R.E. Benner, SIAM Journal on Scientific and Statistical Computing, Vol. 9, #4, July 1988.
- [12] "A Microprocessor-based Hypercube Supercomputer", J.P. Hayes, T. Mudge, Q.F. Stout, IEEE Micro, p6-17, 1986.

- [13] "PROTEAN: Deriving Protein Structure from Constraints", B. Hayes-Roth, B. Buchanan, O. Lichtarge, M. Hewett, R. Altman, J. Brinkley, C. Cornelius, B. Duncan, O. Jardetzky, p417-432 from Blackboard Systems, Edited by R. Engelmore & T. Morgan, 1988.
- [14] "A Computer Program for Generation of Constitutionally Isomeric Structural Formulas" H. Abe, T. Okuyama, I. Fujiwara, S. Sasaki, J. Chem. Inf. Comut. Sci. 1984, 24, 220-229.
- [15] "A Multiprocessor Design in Custom VLSI", D. Jurasek, W. Richardson, D. Wilde, VLSI Systems Design, p26-30, 1986.
- [16] "CSEARCH: A Computer Program for Identification of Organic Compounds and Fully Automated Assignment of Carbon-13 Nuclear Magnetic resonance Spectra", H. Kalchhauser, W. Robien, 1985, 25, 103-108.
- [17] "Combinatorial Problems in Computer Assisted Structural Interpretation of Carbon-13 NMR", A.H. Lipkus, M.E. Munk, J. Chem. Inf. Comput. Sci. 1985, 25, 38-45.
- [18] "Automated Classification of candidate Structures for Computer-Assisted Structure Elucidation", A.H. Lipkus, M.E. Munk, J. Chem. Inf. Comput. Sci. 1988, 28, 9-18.
- [19] "Artificial Intelligence Used for the Interpretation of Combined Spectral Data. 3. Automated Generation of Interpretation Rules for Infrared Spectral Data", H.J. Luinge, G.J. Kleywegt, H.A. Van't Klooster, J.H. Van Der Maas, 1987, 27, 95-99.
- [20] "Structure Generation by Reduction: A New Strategy for Computer-Assisted Structure Elucidation", B.D. Christie, M.E. Munk, J. Chem. Inf. Comput. Sci. 1988, 28, 87-93.
- [21] "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures", P. Nii, The Al Magazine, p38-53, Summer, 1986.

- [22] "Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective", P. Nii, The AI Magazine, p82-106, August, 1986.
- [23] "CHEMICS-F: A Computer Program System for Structure Elucidation of Organic Compounds", S. Sasaki, H. Abe, Y. Hirota, Y. Ishida, Y. Kudo, S. Ochiai, K. Saito, T. Yamasaki, J. Chem. Inf. Comput. Sci. 1978, Vol. 18, #4, 1978, 211-222.
- [24] "Structure Elucidation System Using Structural Information from Multisources: CHEMICS", S. Sasaki, Y. Kudo, J. Chem. Inf. Comput. Sci. 1985, Vol. 25, 252-257.
- [25] "Mapping Parallel Applications to a Hypercube", K. Schwan, W. Bo, N. Bauman, P. Sadayappan, F. Ercal, p141-151, The Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, California, 1988.

Hypercube Expert System Shell Applying Production Parallelism

William A. Harding George A. Sawyer Gary B. Lamont

Department of Electrical and Computer Engineering School of Engineering Air Force Institute of Technology Wright-Patterson AFB, OH 45433

Abstract

Production system implementations of expert systems are becoming more prevalent in a large number of diverse specialties, but the relatively slow execution speed of these systems precludes their use in most realtime applications. The feasibility of improving the performance of production systems software using parallel computer architectures is an area of significant interest in artificial intelligence research. One of the parallel computer architectures of interest is message-passing multicomputers. Production parallelism attempts to apply parallelism at a very high level in order to establish problem granularities that are compatible with current generation multicomputer architectures. Even when using larger grain parallelism, high performance production system shells represent a problem for some multicomputer architectures. The problems related to implementing a production system shell on a messagepassing multicomputer and mapping a specific application to this implementation are analyzed. Further research into the problems of parallelizing real-time production systems applications is also discussed.

Introduction

Intelligent real-time control and real-time monitoring tasks are one of the most challenging new environments for computer applications. Efforts to apply traditional software implementation methods to some of these areas have typically met with failure due to the unmanagability of the associated computer code [15, 77]. Researchers are discovering that complex tasks performed by trained human (like piloting an aircraft) lend themselves to solution using artificial intelligence (AI) approaches. One of the most successful areas of AI research is in the area of expert systems, computer

programs capable of emulating the problem solving capabilities of a human expert in a specific field of knowledge [4, 5]. The most significant problem facing the use of expert systems for real-time tasks is their slow execution speeds. A real-time requirement means "there is a strict limit by which the system must have produced a response to environmental stimuli regardless of the algorithm it employs" [14, 10]. A number of expert systems proposed and developed for real-time use are not capable of sustaining this level of performance [10, 27].

The feasibility of improving the performance of realtime expert systems (particularly those based on the production system paradigm) using parallel computer architectures is an area of current interest in AI research. The performance requirements of real-time productions systems motivates the researcher to eliminate common expert system inefficiencies. These inefficiencies include the overhead associated with symbolic languages (e.g. Lisp, Prolog) and the disproportionately large amount of time spend in the pattern matching phase of a production system's match-selectact cycle. Despite the use of more efficient languages (such as C) and match algorithms (such as Rete), the problem related to match phase efficiency persists [5, 14]. This problem becomes the focus of increasing the execution speed of expert systems through the use of parallel computer architectures.

Background

Robotic Air Vehicle Application

One sponsor of research in this area, the Air Force Wright Aeronautical Laboratories (AFWAL), requires a fast multiprocessor architecture to process an expert system capable of piloting a robotic air vehicle (RAV) [12, 1326]. Under contract with Texas Instruments

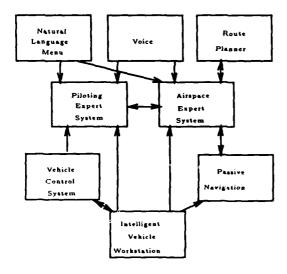


Figure 1: RAV System Architecture [12, 1327]

(TI), the RAV was developed as a production system based expert system using the Automated Reasoning Tool (ART) implemented on several TI Explorer Lisp machines. Figure 1 shows the overall structure of the current RAV implementation including the piloting expert system and airspace expert system. Both the piloting and airspace expert systems applications are based on the production system paradigm. Unfortunately, the RAV expert system has proven to be too compute-intensive to yield results in real-time on any serial or parallel computer architecture (hardware and software) developed to date [2].

Production Systems

The production system paradigm is one of the most common methods for implementing expert systems applications. Production systems apply pattern directed search (inference) using rules which are based on a subset of first order predicate logic. The execution of these rules modifies a set of facts which describe the current state of a specific problem. The following expanded definition describes a production system by its basic components [11, 185]:

- a set of facts, collectively known as working memory (WM) that describe the current state of a problem. A common way of expressing rules is as an object-attribute-value triple [5, 8] such as:

 (autopilot-switch position off)
- a set of rules, collectively known as the rule base.
 Each rule represents an element of problem solving knowledge for a specific application. A Rule typically contains 1 or more condition elements (CEs) (i.e. logical predicates) that must be sat-

isfied in order for them to be applied to solving the problem. A CE can be thought of as an object with an instantiated values, capable of being matched with facts in WM. Rules have the general form:

if (these facts exist) then (execute this action)

- a control structure known as a production cycle that performs the generalized search of the problem state-space:
 - 1. Match evaluate the "if" part of each rule to determine whether it is consistent with the current contents of WM. This is equivalent to producing all possible extensions of the current state of a problem.
 - 2. Select choose one rule from those passing the match test, if no rules are eligible, terminate execution. This is essentially the same as choosing the most promising extension of the current state (path).
 - 3. Act perform the actions specified in the "then" part of the chosen run and return to match phase unless an explicit stop condition exists. This phase updates the current state of the problem (WM), forming a new state.

The Rete Match Algorithm

The match phase of the production cycle accounts for approximately 90% of the total execution time in most production systems [6, 70] and therefore becomes the focus for increasing production system performance though the use of specialized algorithms. Rete is an efficient match algorithm which exploits two prevalent characteristics of most production system applications to obtain significant increases in execution speed: First, in a typical production system application, only a small fraction of the WM changes during each production cycle. Rete takes advantage of these small WM changes by storing CEs satisfied during previous production cycles and using them to aid in satisfying rules during subsequent production cycles. This "state saving" feature means that only recent changes to WM need to be evaluated during the current match phase instead of evaluating the entire WM. Second, Rete exploits the commonality of CEs in the "if" side of the rules in the rule base. By eliminating redundant CEs, the Rete algorithm evaluates a given CE only once each cycle even though this CE may exist in the "if" side of a number of rules. Eliminating redundant CEs tends to significantly reduce the number of pattern matching tests that need to be accomplished during each production cycle [3, 35].

The Rete algorithm uses a special type of constraint network compiled from the "if" side of rules in the rule

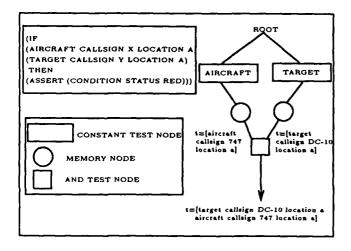


Figure 2: A Single Rule Rete Network

base to perform efficient pattern matching. Figure 2 shows an example of a simple Rete network compiled from a single rule. The Rete network is generated at compile time, before the production system application is actually executed. At run-time, entities (i.e. tokens) containing a flag, a list of time tags and a list of variable bindings flow through the Rete network during each match phase. Each token flows through the network only until a match for its list of variable bindings is not possible. A token reaching the bottom of the graph contains a complete and valid list of variable bindings, making it eligible for execution. [3, 38].

The Rete network contains three basic types of nodes [7, 688]:

- Constant Test Nodes test for consistency between attribute values of individual CEs, such as binding the aircraft objects location attribute value to "a" in figure 2. These nodes appear in the top part of the network and take less than 10% of the total Rete network update time.
- Memory Nodes store the results of previous match phases for use in the current match phase. The state stored by these nodes consists of a list of all previous tokens that match the CEs bound up to this point in the network. This means that only changes made to WM by the most recent rule firing need to be considered during the current cycle.
- Two Input Nodes check consistency of variable binding between two different token inputs.
 A new token is propagated if and only if there is complete consistency between the variable bindings of the two inputs. For example, the "and"

node in figure 2 tests the consistency of binding between the aircraft and target objects. Because the location attribute value of both objects match, a joined token is propagated.

Parallelizing Rete

Extensive research by Gupta shows that the Rete match algorithm is suitable for efficient parallel production system implementations. The data flow nature of the Rete algorithm makes it possible to execute the actions of a number of Rete nodes in parallel. It's also possible to process multiple changes to WM in parallel. The following two general methods are typically considered in parallelizing the Rete algorithm: production parallelism and node parallelism [5, 19].

Production parallelism divides an application's rule base among a number of processors. Each of these processors then performs the match phase on its partition of the complete rule base. Because production parallelism constitutes a static partitioning, its execution involves very little communication between processors. Hence it is a relatively large grain problem. Unfortunately, this static partitioning can lead to large variances in processing time between processors thus limiting achievable speed-up [5, 46].

Node level parallelism attempts to execute the actions of a number of Rete network node in parallel using different processors. Node level parallelism requires a minimum of two communications per node processing action making it a relatively small grain problem. Despite the heavy communication costs associated with node parallelism, it allows for efficient and balanced use of all processors [5, 49].

A significant amount of research has been directed toward implementing Rete on multiprocessors, but the prospects of implementing Rete on multicomputers remains relatively unexplored. Multiprocessors have been favored, because simulation shows node parallelism is superior to production parallelism [5, 55], and the small granularity of node parallelism is not efficiently compatible with the communications facilities of most message passing multicomputers [7, 687]. Limited theoretical analysis of a parallel architecture which implements Rete in an object-oriented manner on a multicomputer has been performed, but it draws heavily on previous multiprocessor design concepts [7, 689].

Research Methodology

It is common practice for parallel computer researchers to compare the performance of their design with current state-of-the-art performance applied to the same problem, however this comparison actually tells us very little about the relative merits of the de-

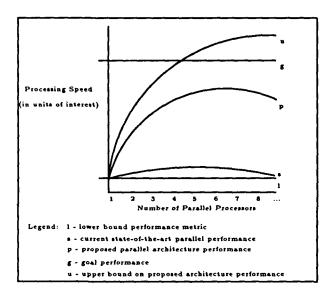


Figure 3: Example Performance Spectrum Chart

sign. The direct comparison of designs is necessary to show advancement of knowledge in the field, but there are two basic reasons why this limited analysis is generally not adequate: First, new parallel designs are often configured to run on different machines where speedups attributable to different hardware cannot be distinguished from speedups attributable solely to new program design. Second, it is not enough when the new parallel design outperforms the existing one, because the performance of both designs may fall far short of the required performance for the proposed application.

In order to determine the true merit of a parallel design (for a given implementation), one must determine where on the performance spectrum the current implementation lies in terms of processing speed. This processing speed is defined in terms that are significant to the particular application (e.g., for expert systems, the performance metric is typically expressed as the average number of rules fired for every second of runtime). By looking at the performance of the current implementation with respect to other important metrics, the actual merit of the current implementation can be seen more clearly.

The performance spectrum encompasses five important measures of performance for determining the relative merits of the design and for guiding the research forward in a logical and methodical manner. Figure 3 shows an example performance spectrum based on the five measures of performance explained below:

1. The state-of-the-art performance (prior to this re-

- search) for this application when implemented on a parallel computer architecture.
- 2. The minimum goal performance value in terms of execution speed required for this application to be feasible. This measure is independent of computer architecture considerations.
- 3. The lower bound performance determined by the processing speed of this application using the best known sequential algorithm implemented on a comparable sequential computer architecture.
- 4. The theoretical upper bound performance for this application based on the selected parallel algorithm and its implementation on a specific parallel computer architecture under ideal conditions.
- The actual measured performance of the new parallel implementation. Ideally, this measurement should be available for different numbers of processors.

This performance spectrum approach results in two important contributions: First, at any point in time, if the research does not appear promising, the researcher has the choice of proceeding with the current design, altering the current design or falling back on another possible design. Second, the performance spectrum indicates whether it is theoretically possible to attain the desired goal performance using the chosen combination of algorithm and architecture. Performance spectrum information also supports intelligent decisions on whether to proceed with refinements of the current design or to try new designs that eliminate weaknesses of the current implementation.

Previous Best Performance

The performance of any new parallel architecture must be compared to that of any existing parallel architecture that is considered state-of-the-art. The only parallel architectures applied to the RAV expert system are those developed by Donald Shakley [15]. This effort concentrated on a Lisp implementation of the RAV on a first generation Intel Hypercube (iPSC1) multicomputer. Using production parallelism and the updated Winston and Horn inference engine, Shakley's iPSC/1 implementation fired an average of 1 rule every 11 seconds with a peak performance of 0.5 rules every second using 16 processors [16, 1352].

Although this study shows that the increased parallelism of the iPSC/1 design could in fact produce processing speedup compared to the TI Explorer design, the amount of speedup realized was hampered by the effects of interprocessor communication overhead, load imbalance, and modelling errors. The program design was implemented in Common Concurrent Lisp

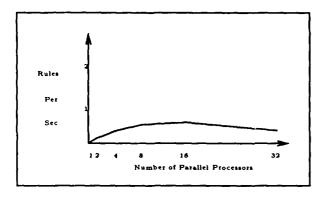


Figure 4: iPSC1 RAV Performance Results

on the iPSC1, which is significantly different than the proposed environment for this research. Without reimplementing Shakely's program on the iPSC/2, a direct comparison with the new parallel design is not possible. Reimplementing the code on the iPSC/2 was not within the scope of this research; however by scaling the previous iPSC/1 results, a more direct comparison is possible. Accounting for the differences in the iPSC/1 and iPSC/2 architectures, 8 rules per second represents an optimistic upper bound on Shakely's implementation on the iPSC/2.

Goal Performance

The goal performance for a real-time application determines how "fast" a given design must be in order to meet the feasibility requirements of the application, and this figure in turn influences the design of possible solutions. AFWAL managers were interviewed to determine what the best estimate of the real-time requirements for the RAV system are and the critical points of the RAV mission are. The estimated goal performance for the RAV application was determined by the rate of the incoming data and the number of rules needed to process this data and provide control outputs before the next set of data is received. The most critical parts of the RAV application involve lowlevel route following and final approach to landing. During these phases of the RAV mission, the system design must be capable of sustained firing of between 37 and 75 rules every second [2].

Lower Bound Performance

A "good" parallel implementation typically starts with a "best" or at least "good" serial implementation in terms of algorithms and data structures. Parallelizing a less than optimum serial algorithm is typically justified only when the optimum algorithm is not amenable to parallelism. If a less than optimum algorithm is selected, increased performance through parallelism is required just to obtain the optimum serial algorithm's level of performance [8, 122]. For this research, the processing speed of a "good" serial design is needed to delineate a lower bound on performance expected in processing the RAV expert system. Furthermore, the serial design must be implemented on hardware similar to the hardware upon which the subsequent parallel designs are implemented if the comparisons are to be valid.

The proposed serial design employs an existing expert system inference engine or shell, which uses the Rete algorithm when performing the match phase of the match-select-act cycle. The possible alternatives viewed were: Inference Corporation's Automated Reasoning Tool (ART), Official Production System, version 5 (OPS5) from Carnegie-Mellon University and the NASA developed C-Language Integrated Production System (CLIPS). The original RAV system is implemented using ART, however ART is available only in Lisp and Bliss based versions. Sequential versions of OPS5 have the same problem, but Carnegie-Mellon has recently developed more efficient versions of OPS5 expressly for parallel applications. Unfortunately, the parallel version of OPS5 uses a combination of C and machine language making the shell non-transportable. NASA's CLIPS interpreter is the expert system shell chosen for this research. The C language composition of CLIPS supports transportability and its syntax is similar to ART making transliteration of application rule bases easier.

The lower bound performance implementation uses a full CLIPS interpreter executing the RAV rule base on the host processor of the iPSC/2. At system initialization, the processor is loaded with the entire RAV rule base (326 rules) and a set of facts describing the initial state (389 facts). In AFWAL's implementation of the complete RAV system, several conventionally programmed subsystems provide continuous simulated inputs to the piloting expert system subsystem. Because these systems providing inputs to the RAV expert system were not available under this design, another approach to providing these inputs was required. The "if" side of rules involving processing of input data were modified to obtain data from WM instead of from outside systems. This data was provided by a set of data facts asserted along with the initial facts. This study's benchmark test simulates a limited RAV mission; in this test, a total of 73 rules are fired as the RAV progresses through the simulated mission. Over numerous test runs, the sequential implementation averaged 3.5 seconds to fire the 73 rule benchmark test. The average 20.9 rule per second figure now becomes the lower bound performance.

Upper Bound Performance

Certain assumptions are made at the outset of this analysis to model an ideal environment for the RAV expert system design executing on the proposed parallel architecture. First, the model assumes perfect load balance among the available processors and that no computational overhead is introduced through parallelism. Second, the model assumes that the only activity other than computation on a processor that produces time cost is inter-processor communication. Under the HyperCLIPS design, the execution time for a single match-select-act cycle is:

- The maximum time spent by any given processor to update its local Rete network and select its "best" local rule, plus
- the time for the processors to determine which processor has the global "best" rule through graycode compare/exchange (with its neighbor processors), plus
- the time for the processor with the global "best" rule to broadcast the actions associated with the best rule to all processors.

Given the time components related to a single match-select-act cycle, it is possible to calculate upper bound performances for the complete RAV benchmark test. Under the assumptions of the model (even load balance and parallelization overhead), the total execution time should be evenly divided by the number of processors. This parallel computation model represents a linear decrease in processing time with respect to the number of processors used. The easiest method for computing the total minimum communication time involves finding the minimum communication time per cycle and then multiplying by the number of cycles. By adding this minimum total communication cost to the linear speedup figure representing the total computation time, the total minimum computation time can be determined. Using N processors running a restricted rule base with C cycles, the upper bound performance of the proposed design, in seconds can be expressed as:

$$t_{PAR} = \frac{t_{SEQ}}{N} + t_{SE} + t_{AB}$$

where t_{PAR} is the parallel computation time, t_{SEQ} is the sequential computation time, t_{SE} is the select-exchange time and t_{AB} is the act-broadcast time.

The task of determining the actual upper bound performance now becomes that of acquiring actual times for the variables in the previous upper bound equation. This process can be significantly simplified if we consider the computation and communication costs separately. Finding the minimum computation time is a relatively straightforward task. From the lower bound

performance measurements, we know that the serial implementation of CLIPS requires 3.5 seconds to execute the 73 cycle restricted rule base. The minimum parallel computation time now becomes the serial processing time divided by the number of processors used.

In comparison with the computation time, determining the minimum communication time is a somewhat more involved task, but provides a more accurate indication of the actual cost. A total of d+1 communications between processors is required to determine which processor has the best candidate rule, where d is the dimension of the cube being used. Only one broadcast communication is required for the processor with the "best" global rule to send the actions associated with that rule to all other processors.

Using a simple message passing ring program, it was determined that a compare and exchange communication requires 0.00424 seconds and the average action message broadcast requires 0.00776 seconds [9, 6-5]. The iPSC/2's 2.8 Mbytes per second interconnection network is capable of transferring each compare and exchange communication in 0.0000825 seconds and the broadcast message in 0.0001678 seconds. These times are two orders of magnitude less than the message transmission times actually observed. It becomes obvious that overheads associated with preparing the data for transmission, setting up the message transmission path and converting the message to data at the receiving node account for most of the actual message passing time.

Given the lower bound processing time and communication time figures, the upper bound performance (in seconds) can be expressed as:

$$(\frac{3.5}{N}) + ((((d+1) \times 0.00424) + 0.00776) \times 73)$$

where N equals the number of processors and $N = 2^d$. Figure 6 shows the results of this analysis applied to different numbers of processors.

HyperCLIPS Design and Performance

The HyperCLIPS system design implements a parallel production system interpreter in CLIPS using the production parallelism concept. This design supports the use of parallelism in all three phases of the match-select-act cycle. A general high level description of the HyperCLIPS algorithm follows:

- 1. While termination is not detected, continue
- 2. Parallel Match
 - each processor receives WM changes from a root processor
 - each processor updates its local Rete network based on WM changes

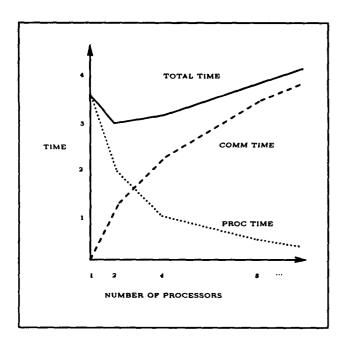


Figure 5: Upper Bound Performance Metric

3. Parallel Local Select

 each processor selects the "best" rule from its local conflict set

4. Global Select

- processors perform compare/exchange of local "best" rule priority with neighboring processors
- root processor holds the "best" global rule when compare/exchange is complete

5. Broadcast Global Act

root processor broadcasts WM change specified by the global "best" rule's "then" side.

6. Return to step 1

Under the concept of the HyperCLIPS design, each active processor supports a full production system interpreter; each processor executes this CLIPS shell program on a subset of the total rule base. At system initialization, the iPSC2 host processor loads all working processors with an approximately equal subset of the rule base. This research makes no attempt to allocate rules in an optimum manner with respect to load balance among processors. Instead, an adhoc allocation is used to distribute the rules evenly among processors. With this static decomposition approach, no interprocessor communication is required

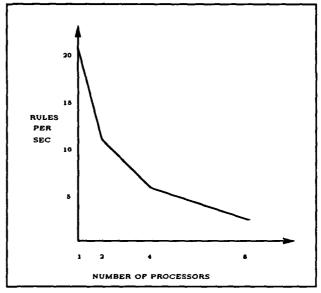


Figure 6: Experimental Performance (RAV)

during the match phase. Each processor's local match phase requires no communication because all information needed to update the Rete network is already local to that processor.

The most significant modification to the serial CLIPS code involves adding the message passing capabilities between processors. For the compare/exchange communication, each processor exchanges messages with all its nearest neighbor processors in the binary N-cube network. This message contains only the priority of a given processor's "best" rule. The processor with the highest priority rule becomes the root processor and broadcasts the action associated with its chosen rule to all other processors. This communication is simply an ASCII string that is processed by the CLIPS interpreter on each processor.

The upper bound prediction, shown in figure 5, indicates that almost no speed-up can be expected by this design and indeed, the results are somewhat discouraging. Figure 6 shows that, as the number of processors increases, the amount of actual speedup actually decreases. On the surface, it appears that communication overhead is the culprit, but further analysis shows it to be only a contributing factor. The communications costs shown in figure 5 represents an accurate measurement of the average communication times whereas the computation times are optimistic. This knowledge allows us to make a meaningful assessment of the load balance among processors achieved in this research. Comparing the upper bound results with

those in figure 6, it is obvious that the load balance among processors is far from optimum. This problem raises the question of whether an efficient partitioning of the RAV expert system is possible.

Performance Comparison Findings

This research indicates that the basic methods behind this design are promising, but the design and implementation suffer due to influences of the parallel architecture chosen and the characteristics of the application itself. The lower bound performance experienced by the serial CLIPS design is impressive (especially considering it is implemented on a micro-computer), but it still falls short of the minimum real-time requirements. CLIPS performs particularly well because it takes advantage of the Rete state-saving algorithm for the match phase of the match-select-act cycle. This is largely the reason that the serial version outperforms Shakley's parallel implementation [15, 62].

The results show that the HyperCLIPS implementation on the iPSC/2 will not produce effective results for the RAV application. The upper bound on performance indicates that almost no speed-up is achievable using the HyperCLIPS design due primarily to large communication overheads. Even when using minimum communication between processors, the overhead incurred constrains speedup to 1.13 times. In the case of the RAV application, it is both load imbalance and communication overhead that conspire to produce negative speedups. The ad-hoc partitioning method used for the RAV application fails to effectively balance the processing load. The results indicate that using multiple processors and ad-hoc rule partitioning does not reduce the serial processing time. These results warrant an investigation of the RAV expert system characteristics to determine why the use of multiple processors does not appear to decrease execution time. Additionally, the use of some algorithmic mechanism to partition rules among processor for effective load balance must be investigated.

The Partitioning Problem

The characteristics of expert systems applications becomes significant when considering the optimum manner for parallelizing them. Empirical measurements by Gupta and Forgy uncover two of the more vital characteristics of production systems [6, 93]:

- The affect-set (the set of rules affected by a given rule firing) is generally very small with respect to the total number of rules in the application.
- The size of the affect-set does not increase as the number of rules in the application increases, instead it remains approximately constant (between 25 to 40 for the systems measured).

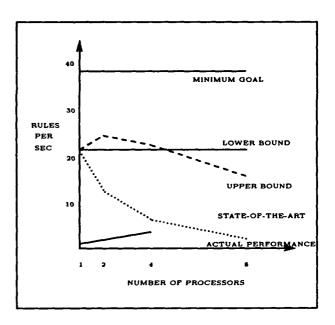


Figure 7: Actual Performance Spectrum (RAV)

Gupta discusses the affect-set problem and gives several plausible reasons why this problem exists in production systems. If we subscribe to Gupta's assertions that small affect-sets are basically an inherent part of production systems, then production parallelism becomes significantly less attractive as the following section discusses.

Production parallelism relies heavily on the existence of fortuitous rule-to-processor allocations in a given application. In order for production parallelism to produce significant speedups, the computational load associated with the match phase must be as evenly distributed as possible. Considering the Rete algorithm, this means that the computational load of updating the local Rete network on each given processor must be as equal as possible. Oflazer reminds us that this problem must be considered for the case of firing every rule in the rule base, particularly those rules that fire more often than others [13, 96]. Oflazer proves that this problem is NP-Complete, but also describes an efficient heuristic approach for rule partitioning [13, 94]. More recently, Dixit and Moldovan describe a method for allocating rules to processors that is independent of Rete-based implementations. Their motivation is based on finding more generalized ways of expressing and applying parallelism in production systems that is independent of algorithm-specific details [1, 24].

Offazer's partitioning algorithm results in rule-toprocessor allocations that produce 1.15 to 1.25 more speedup than ad-hoc allocation methods [5, 155]. There are several reasons that this partitioning does not produce better results [5, 111]:

- 1. The size of the affect-set constrains the number of processors that can produce effective work. If the number of processors is larger than the size of the affect set, then the there will be some processors that do not contain rules affected by the firing of a given rule. Whenever this situation occurs, the processors without affected rules will be essentially idle.
- The time to process different rules in the affect-set can vary greatly depending on the specific application. If each processor has one affected rule, then the actual time each processor takes to update that rule will produce some computational load imbalance.
- 3. Loss in Rete network sharing will tend to increase the number of redundant computations. Rules in the affect-set are likely to contain at least some common condition elements. If these rules are allocated to different processors, then each processor will perform redundant computations while updating its local Rete network.

Top-level examination of the RAV expert system indicates that it is not particularly amenable to production parallelism methods. Analysis indicates that each rule firings affects an average of just four rules, with an observed range of between 1 and 18. Four rules represents only 1.5% of the 273 rules in the RAV rule base. These results indicate that the typical affect-set for the RAV expert system is significantly smaller than the applications measured by Gupta and Forgy. Variations in rule processing time within the affect-set and loss in Rete network sharing among processors have not been examined. The existance of small affect-set sizes is sufficient to explore the assertion that the RAV application is not amenable to production parallelism.

This discussion indicates that production parallelism is significantly constrained by the characteristics of the particular application, particularly the size of the affect-set [5, 113]. The average affect-set size is a reasonable determinant of the upper-bound speedup that can be expected using production parallelism if processing differentials and losses in Rete network sharing are not considered. The size of the average RAV expert system affect-set (four), now becomes the upper bound on the number of processors that can be effectively used to increase execution speed and the upper bound on speedup as well. Other factors not considered in this analysis may influence the upper bound of four somewhat, but they are not likely

to influence it considerably. Other factors aside, it now becomes obvious that production parallelism is only a marginally promising means for increasing the execution speed of the RAV expert system application in its current form due to the struture of its rule base.

Conclusions

Parallel processing is a promising approach to achieving real-time processing of expert systems software, but a number of impressive problems exist between this concept and its implementation. The primary problems that need to be overcome are interprocessor communications overhead and load balancing. The major factor in minimizing both of these problems involves the proper choice of problem decomposition and the parallel computer architecture. Although the Rete match algorithm produces impressive serial performance in processing expert systems, it also results in a problem granularity that is marginally compatible with the architecture chosen for this research. The results indicate that this type of problem requires the use of parallel computer architectures with significantly less communication overhead.

This research goes beyond just producing a new parallel architecture application. Although the performance of the HyperCLIPS design on the iPSC2 Hypercube was less than impressive, it's performance is quantified in terms of the lower and upper performance bound and the ultimate goal performance. This approach not only adds validity to the design, but exposes the level of maturity the RAV expert system research achieves from this research. Based on this approach, the findings indicate that research into parallel processing of the RAV expert system is still in its infancy. The processing speeds obtained from using serial CLIPS supports the continued use of the Rete match algorithm. But, the characteristics of the RAV expert system suggest that the system lends itself to very limited speedup using production parallelism. Therefore, RAV expert system research may be better served by approaching parallelism from the standpoint of node parallelism rather than production parallelism.

The problems encountered in this research underscore the need for significantly more background research in the area of real-time expert systems. Most previous research in applying parallelism to production systems concentrates on "stationary" types of problems instead of real-time types of problems. Processing speed is important in a number of computationally intensive "stationary" applications, but the need for additional processing speed in real-time systems is critical. Laffey notes that Gupta and Forgy's assertions about the level of parallelism in the average production system does not typically apply to real-time

production systems. Real-time applications typically involve changes to WM each cycle as a direct result of incoming data. This characteristic of real-time applications has the potential to increase achievable speedups significantly [10, 40].

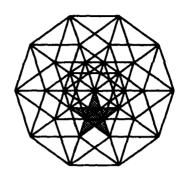
In their article, Fast is not Real-Time, O'Reilly and Cromarty state that guaranteed response time is just important as fast processing capability in achieving real-time behavior. Without guaranteed response times, the system may not be capable of responding quickly enough in critical situations even though its average processing time is very fast [14, 249]. The implication is that average processing times mean very little in terms of real-time expert system performance. Instead, the critical measure now becomes: the maximum time required to process updates to WM at any given time. This problem is likely to be compounded by the assynchronous flow of data into the WM. Laffey even cites research by Halley indicating that Rete is not appropriate for real-time applications because an upper bound on the Rete network update times cannot be accurately predicted [10, 40].

What level of parallelism does exist within real-time expert systems applications? Can this parallelism be successfully extracted allowing significant increases in speed-up using parallel processing methods? Is it possible to accurately predict the guaranteed response time in these systems? Future research will concentrate on determining the feasibility of applying different levels of parallelism to solving the problems inherent in real-time expert system applications.

References

- [1] Dixit, V. V. and D. I. Moldovan. The Allocation Problem in Parallel Production Systems. *Journal* of Parallel and Distributed Computing, 8(1):20-29, January 1990.
- [2] Fanning, 1Lt Jesse. AFWAL Robotic Air Vehicle (RAV) Project Member. Telephone interview. Air Force Wright Aeronautics Laboratory, Wright-Patterson AFB, OH. 22 November 1989.
- [3] Forgy, Charles L. Rete: A Fast Match Algorithm. AI Expert, pages 34-40, January 1987.
- [4] Giarantano, Joseph C. and Gary Riley. Expert Systems: Principles and Programming. PWS-Kent Publishing Co., Boston MA, 1989.
- [5] Gupta, Anoop. Parallelism in production systems. Master's thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, March 1986.

- [6] Gupta, Anoop and Charles L. Forgy. Static and Run-Time Characteristics of OPS5 Production Systems. Journal of Parallel and Distributed Computing, 8(1):20-29, January 1990.
- [7] Gupta, Anoop and Milind Tambe. Suitability of message passing computers for implementing production systems. In Proceedings of the National Conference on Artificial Intelligence, pages 687-692, August 1988.
- [8] Gupta, Anoop, and others. Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis. International Journal of Parallel Programming, 17(2):95-124, April 1988.
- [9] Harding, Capt William A. Hypercube expert system shell - applying production parallelism. Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1989.
- [10] Laffey, Thomas J. and others. Real-Time Knolwedge Based Systems. AI Magazine, 9(1):27-45, Spring 1988.
- [11] Luger, George F. and William A. Stubblefield. Artificial Intelligence and the Design of Expert Systems. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.
- [12] McNulty, Christa. Knowledge engineering for piloting expert system. In Proceedings of the IEEE National Aerospace and Electronics Conference, pages 1326-1330. IEEE, May 1987.
- [13] Oflazer, Kemal. Partitioning in Parallel Processing of Production Systems. *IEEE*, pages 92-100, 1984.
- [14] O'Reilly, Cindy and Andrew S. Cromarty. Fast is not real-time: Designing effective real-time ai systems. In Proceedings of the International Conference for Optical Engineering, pages 249-257. SPIE, April 1985.
- [15] Shakley, Capt Donald J. Parallel Artificial Intelligence Search Techniques for Real-Time Applications. Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, December 1987.
- [16] Shakley, Donald J. and Gary B. Lamont. Parallel Artificial Intelligence Search Techniques for Real-Time Applications. In Proceedings of the Third Annual Conference on Hypercube Concurrent Computers and Applications, pages 1352-1359. ACM Press, January 1988.



The Fifth Distributed Memory Computing Conference

2: Alternate Applications

Parallel Distributed-Memory Implementation of the Corrective Switching Problem

J-Y. BLANC and D. TRYSTRAM LMC-IMAG 46 avenue F. Viallet 38031 Grenoble Cedex (FRANCE) J.W.A. RYCKBOSCH
EDF-DER
1 avenue De Gaulle
92141 Clamart Cedex (FRANCE)

Abstract.

For the past 20 years, an increasing interest has been devoted to the sequential Conjugate Gradient Method for solving large linear systems arising from the modeling of physical problems (especially for very large systems with sparse matrices). This paper deals with the implementation on parallel supercomputers of a preconditioned conjugate gradient method for solving the corrective switching problem obtained while modeling the behavior of power systems in electrical networks. This problem consists in finding the successive solutions of many close linear systems (not too large) with very ill-conditioned matrices (sometimes even singular). We present a new method based on the Preconditioned Conjugate Gradient algorithm with an original preconditioning and study its parallelization on both shared and distributed memory computers.

1. Setting of the problem

During the control of electrical networks, the operator must ensure the system to be in a safe state (i.e. to be able to protect the system against incidents liable to occur in real time). The demand and the possibility of the plants are such that nuclear energy between two plants flows from various nodes of the network. The loss of one element could jeopardize the security of the whole system by a chain tripping: in such case, an overload line occurs and without any operation the protective devices will act and the line will trip out. In actual operations conditions, the switching actions that the operator applies to the electrical network ensure that overloads will disappear before the delayed protective devices go into action. Such actions are shown on the picture at the end of the paper. The computation of switching actions is a combinatorial problem, very hard to solve. The connections of the switching elements are described as discrete variables. The corrective switching problem corresponds to determine the various possible solutions of the load flow calculation. Each such situation requires to solve a linear system where the matrices have only a few elements which differ from each other.

Let us consider the N consecutive linear systems below: (S_i) $A_i x_i = b_i$, $1 \le i \le N$ where the matrices A_i (of size n by n) are "close" to each other, viz, $A_{i+1} = A_i + \Delta_i$, with Δ_i of small norm. The solutions x_i will be close to each other in this sense, and we want to take full advantage of this.

Note that this problem also occurs in Adaptive Filtering or Finite Element modeling.

2. Solving the corrective switching problem

The method commonly used for solving this problem consists of refactorizing the matrix of each system (S_i) by the direct Cholesky Method and solving it separetely. Note that the use of this method is not available when n becomes too large, for both reasons of huge storage and high rounding errors. Practically speaking, n is about 100 for a typical corrective switching problem. However, during the parallelization, the successive solutions can be obtained simultaneously on multiple processors without any communications, and the local computations require a load-balanced amount of operations, namely O(n³). Moreover, the solution will be difficult to carry out because the matrices are very ill-conditioned (they can even be singular, that means in practice that we obtain several solutions).

3. A better algorithm based on the Conjugate Gradient

A new iterative method based on the Preconditioned Conjugate Gradient Method has been proposed by the authors for sequential computers. It takes into account the small norm variations of the matrices. The basic idea is to factorize a matrix A_i (by Cholesky) for a given index i and to use it as a preconditioning of the A_j (for j>i) until this factorization differs too much from A_j . We can typically solve about 50 linear systems with the same preconditioning.

A complexity study has proved the superiority of this method in regard to the usual methods, since each step costs less than a Cholesky factorization and the preconditioning is not computed at each step, but only for the next reinitialization.

The figure below gives the numerical results on an usual sequential computer (SUN3).

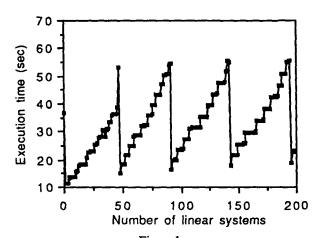


Figure 1.
Successive Preconditioned Conjugate Gradient method.

It is well-known that the Preconditioned Conjugate Gradient algorithm gives the solution of a system of size n in less than n iterations. Each iteration requires, refering to the basic linear algebra subroutines BLAS1 (vector level operations), two DOTs (inner products) and three SAXPYs (vector updates), plus the solving of a system (usually not too expensive to solve) and a matrix-vector product (both needing O(n²) operations), which are BLAS2 operations and can be also decomposed into BLAS1 vector elementary operations, and the evaluation of two scalar parameters. Because of both the data and operations regularities, the granularity of the tasks will be taken as O(n) and are suitable for an implementation on vector processing units.

This analysis leads to two basic ways to implement the successive Conjugate Gradient in parallel: first, we can run each Conjugate Gradient algorithm locally to the processors (this first solution will be limited by the local memory size), or we can parallelize successively the most expensive task of one Conjugate Gradient (the BLAS2 operations) to run it concurrently on all the processors.

4. Parallel implementation on shared-memory computers

The parallelization of numerical algorithms on this kind of parallel computers has been much studied. It is quite simple if we take into account the analysis of the precedence constraints. The schedule of the tasks is synchronized even if the numbers of iterations are different on the various processors. The large shared-memory allows every processors to get easily all the global informations (like the knowledge of the Cholesky factors used as preconditionings for several systems). The use of local cache memories speeds-up the execution time after the duplication of the common data.

The strategy consists in dispatching the various systems to the processors. Note that the shared-memory vector-computers are limited by the slight number of processors (no more than 10 in practice).

5. Parallel implementation on distributedmemory computers

To find an efficient implementation of this method is difficult because successive sytems are solved with the same preconditioning and we need a global checking to ensure that some processors are not locally computing too many iterations. Thus, the amount of computations is not well balanced from a processor to the other. In order to simplify the computation, an initial phase is run where the number of linear systems to be solved with the same preconditioning will be determined. This initialization leads to a static tasks allocation.

We propose here an implementation where a processor computes first the Cholesky factor of A₁ and then broadcasts it to the others. Each processor computes the solution of a system with a given amount of computations (that means a given number of iterations, which increases with the index i) inversely proportional to its distance from the sender processor.

Then, we asynchreonously compute and broadcast a new Cholesky factor to the processors. Thus, the first processor to receive the data is the one which has to do the most iterations. The aim is then to find the "better" allocation for the successive linear systems to the processors, in fact the one which minimizes the total execution time.

The following figure gives the numerical results for the same example as before for a Cholesky factorization plus the resolution by successive Conjugate Gradient method until the next reinitialization.

The experiments have been performed on a 32 hypercube vector computer (FPS T40). The various results represent the various levels of programming of each processor.

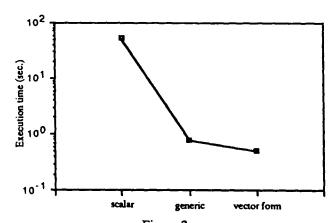


Figure 2.
Schedule of work on the distributed-memory vector-computer

Another way to implement the corrective switching problem is to distribute among the various processors all the local informations about the physical problem. This approach will completely change the problem to solve: instead of having a linear system, each processor should exchange local informations between neighbor processors, perform local elementary operations such as the sum of the electric powers stemming from a node in all the directions.

6. Conclusion

We conclude by numerical comparisons between shared-memory and distributed-memory computers (of the "same magnitude order" of performances). Numerical experiments have been run first on a shared-memory parallel computer (Alliant FX80 with 8 vector-processors of each 16 MFLOPS of peak performance) and a distributed-memory parallel computer (FPS T40 hypercube with 32 vector-processors of each 12 MFLOPS of peak performance).

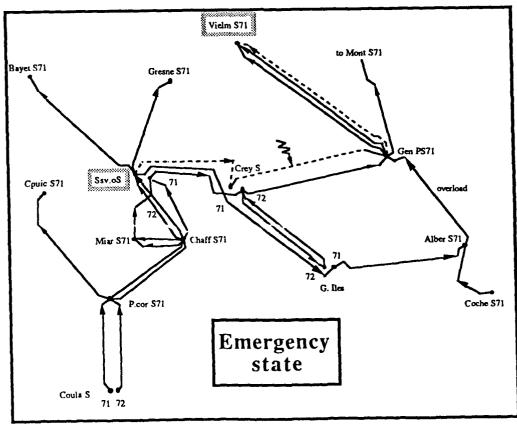
The experiments show the good behavior of the successive preconditioned conjugate gradient method for solving the practical corrective switching problem on parallel computers. The distributed-memory implementation is better because of the larger number of processors and worst as the shared-memory implementation for the same number of processors, as it was expected.

Basic References.

J-Y. BLANC, P. COMON and D. TRYSTRAM, Using Preconditioned Conjugate Gradient for Consecutive Linear Systems, Comm. in Applied Numerical Methods (to appear) Z. CVETANOVIC, The Effect of Problem Partitioning, Allocation and Granularity on the Performance of Multiple-Processor Systems, IEEE TC, Vol. 36, N. 4, 1987 J.J. DONGARRA, F.G. GUSTAVSON AND A. KARP, Implementing linear algebra algorithms for denses matrices on a vector pipeline machine, SIAM Review 26, 1984 G.H. GOLUB and C.F. VAN LOAN, Matrix Computations, Johns Hopkins University Press, 1983 J. RYCKBOSCH, A Method for Solving the Problem of Optimal Swithching, proceedings of IFAC, Pekin, 1986

Example of a practical corrective switching problem.

We give below the picture of a fragment of the Electrical High Voltage (EVH) French system in a very strained situation. The demand and the possibility of the plants are such that nuclear energy of Bugey and Cruas plants flow from Ssv.OS71 to Vielm S71. The loss of one of the Crey-Gen element could jeopardize the security of the whole system by a chain tripping: in such case, an overload of the Alber-Gen line occurs and without any operation the protective devices will act and the line will trip out.



Fault Simulation on Message Passing Parallel Processors

Leendert Huisman
Indira Nair
I.B.M. Thomas J. Watson Research Center

Raja Daoud
The Ohio State University

Abstract

A new parallelization technique for Fault Simulation is described that is suited for message passing based parallel processors. The problem is parallelized by first casting it in Dataflow form and then constructing a Dataflow emulator for message passing systems. A fault simulator for combinational logic has been implemented on a Transputer based parallel processor, the IBM VICTOR multiprocessor. Overall performance has been measured for several logic designs.

1. Introduction

Fault simulation [1] means the simulation of a logic design that has been modified to reflect the presence of a fault. Simulating such faulty designs is done, among other things, to asses the ability of a proposed set of test patterns to expose faults in the real design. Typical faults that are simulated are any input pin or output pin of any gate stuck at 1 or stuck at 0. In principle, each such fault gives rise to a modified design that has to be simulated. Many such faults are equivalent however, in the sense that the corresponding modified designs behave identically. Typically there are on the order of 3 or 4 non-equivalent faults per gate in a logic design. Simulating all these modified designs is therefore very costly.

In developing our parallel fault simulation algorithm, we considered various attributes that the fault simulator should have. Not all attributes have been implemented yet, but we feel confident that they can, with the approach that we have used.

How these attributes are implemented depends on the hardware characteristics of the parallel processor. The parallel processors that we will consider in this paper are distributed memory machines. Such a machine consists of a number of nodes, also called processors. Each node has a CPU and some local memory, typically in the order of several megabytes. The nodes communicate by sending messages to each other.

First, we wanted the simulator to be flexible. By this we mean that it should not restrict too much the range of logic designs that can be simulated. For example, the simulator should be able to handle easily very large designs, designs with embedded memory elements like latches and feedbacks. Parallel pattern techniques[2-5] do not have this attribute, because they do not handle memory elements or feedbacks efficiently. The inability to handle feedbacks well also excludes pipeline techniques[6, 7].

Secondly, the limit on the parallelizibility of a given problem should be determined by the properties of that problem and not by hardware or software. For example, parallelization according to the single-controller/many-slaves model does not have this characteristic, because for sufficiently many processors the central controller becomes the bottleneck. The maximum speedup is then determined by how fast the central controller can work and not by the degree of parallelizibility of the problem.

 Λ practical measure of how well a problem has been parallelized is the number of processors P_m at which the speedup curve flattens out. Λ good parallelization is one where P_m depends only on some overall characteristic of the problem, like its size for example.

Thirdly, when the number of processors increases, the total available memory increases with it and the size of the largest problem that can be handled should increase as well. It does not always work out that way however, because of the limited local memory that is available to each processor: the partitioning that was used may lead to an overflow of that local memory and if no provisions are made to use memory available on other processors, even small problems may not be handled.

Good memory usage can be described as follows. Assume that the uni-processor on which we run the uni-processor version of our algorithm has infinite memory and let the memory required by this sequential version for a given problem be M. Assume there are P processors, each with local memory of size m. Finally, let the total memory, summed over all P processors, that is needed by the parallel algorithm for the same problem be Mp. Mp should be almost independent of P. Complete independence is not possible, because for example some information about the network connectivity has to be stored somewhere, and this information does increase with P. Memory is then used properly when M_P is bounded by cM, where c is a number that depends only weakly on P and is close to 1 for small P, and when any problem that can run on a uni-processor with memory Pm/c is also guaranteed to run on the parallel processor.

The classical way of parallelizing fault simulation is by partitioning the fault-list among the available processors [8-10]. This parallelization is straightforward because different faults can be handled independently. It is very suitable for shared memory machines where the partitioning of the fault list can be done dynamically [8]. It has the disadvantage on distributed memory machines that the complete design has to be replicated on all processors. Better partitioning algorithms have been designed [11] in which each processor only needs the description of a portion of the design, but even then the total amount of memory required for the design description is considerably more than on a uni-processor.

Finally, all processors should be used as much as possible within the limits set by the inherent parallelizibility of the problem itself. Roughly speaking, this means that most of the time most of the processors should be doing useful work. If the number of processors is P and the total lapse time is t, then the total time taken by the problem is Pt. The total CPU time spent on the problem is called T_{CPU} and is obtained by adding up all the time periods on all processors when a processor is working. The (implementation of the) algorithm is called load-balanced when Pt is not much larger than T_{CPU} . It is

an efficient parallelization when T_{CPU} is not much larger than the CPU time needed by the corresponding sequential version of the algorithm on a single processor.

These requirements lead to some important conclusions. First, when we partition the problem among the available processors, the partitioning should be done such that there is no duplication of design descriptions and no duplication of calculations, to avoid underutilizing memory and/or processing This requirement therefore excludes partitioning the fault list among the processors, because such a partitioning requires multiple copies of the design description and repeated simulation of the same patterns on the fault free design. In addition, if we want to be able to shift jobs from one processor to another, to balance the load among the processors, the jobs should be fairly small. This excludes coarse-grained parallelization like the partitioning suggested in [11].

2. Parallelizing Fault Simulation

A general fault simulation algorithm has three DO loops: one over the patterns that have to be simulated, one over the faults in the fault-list and one over the gates in the design. The loop over the patterns is done in the order in which the patterns are applied to the real design, while the loop over the faults can be done in any order. The loop over the gates is done in topological order [12]. This is defined even for sequential designs because when the real design is tested it is put first in test mode [13]. In this mode, all memory elements are controllable and observable, and the design is reduced to a collection of disconnected pieces of combinational logic. In each such piece, there is a trivial partial ordering among the gates: gate A precedes gate B when B is in the downcone of A.

The fault simulation algorithm that we will consider in this article is (a slightly modified version of) Concurrent Fault Simulation [14]. The overall structure of this algorithm is shown in table 1. The calculation at each gate determines which faults produce fault-effects on the output of that gate for the pattern being simulated. Implicitly, we do a loop over all faults in the fault-list, as indicated in the program fragment, but explicitly we only consider faults that have fault-effects on the inputs of the gate or that are located on the input or output pins of the gate.

```
DO all patterns;
DO all gates in topological order;
DO all faults;
code;
END;
END;
END;
```

Table 1: DO loop structure of Concurrent Fault Simulation

The innermost DO loop, the one over the faults, is treated as an unbreakable, atomic unit. It is the job unit out of which the parallel fault simulation will be built up. This partitioning meets the various requirements mentioned in the previous section. partitioning the problem into disjoint jobs, there is no duplication of design descriptions and no duplication of calculations. In addition, the atomic jobs are sufficiently small that they can be moved around easily when required to maintain load balance. The output of the job is a fault table, consisting of all the faults that produce fault-effects on the output of the gate. Their sizes range from 1 to several thousand in the designs that we considered. When the node where the output table is calculated does not have enough memory to store the table, no local memory overflow need to occur, because the table can be stored on another node.

Each gate needs fault tables from its preceding gates and the corresponding job can therefore not be executed before all jobs corresponding to preceding gates have been executed. Gates will be called independent when they are not in each other's downcone. If two gates are independent, neither has to wait for the other and both can be executed in parallel. It is this parallelism that we want to exploit in our parallel fault simulation. A rough measure of the number of gates that can be treated in parallel is given by the width of the design, i.e. the ratio of the total number of gates in the design and the average number of gates between a controllable input, like a latch or a PI, and an observable output, like another latch or a PO.

3. System Hardware and Software

The parallel Fault Simulation algorithm has been implemented on the IBM VICTOR V256 multiprocessor [15]. This is a Transputer [16, 17] based message passing machine. V256 has 256 nodes, each consisting of a model T800 Transputer [18] and 4 Mbyte of local memory. Communication between

nodes is done via message passing, for which the Transputer provides substantial hardware support. Each node has four links to neighbors and the nodes are connected in a mesh topology. The bandwidth over the hard links between neighbors is on the order of 1.5 Mbyte/sec. More details about the hardware can be found in reference [15].

V256 is connected to a host, a PC/AT. The host has an additional Transputer on which the host program runs. This host program supervises the fault simulation: it requests information from the user: which design to simulate, how many patterns and which patterns, and where to store the results. Loading the simulation programs and the design descriptions onto the nodes of V256 is done from the host as well.

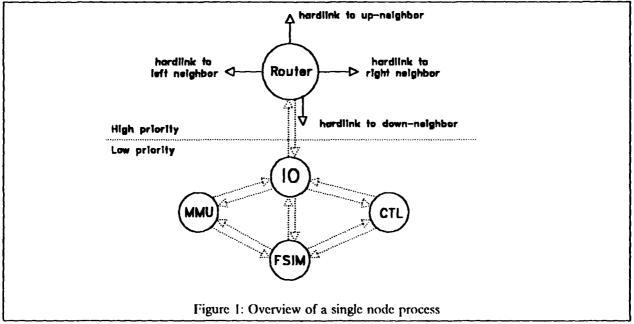
All programs are written in OCCAM [19, 20], a parallel programming language that implements CSP[21]. All nodes run the same program, an overview of which is shown in Figure 1. The program is divided in two processes, that run in parallel but with different priorities (PRI PAR in OCCAM language). The high priority process, above the dashed line, is a router. The low priority process is below the dashed line, and consists of four processes that run in parallel. The actual application program consists of the FSIM and CTL processes. MMU is a memory manager and IO is an input/output interface between the router and the other low priority processes. The five different main processes shown in the figure communicate via soft channels, indicated by the dashed arrows.

The function of the IO process is to funnel messages from the application and the memory manager to off-node processes through the on-node router and to distribute messages from off-node processes between the memory manager and the application. In addition, the IO process plays a crucial role in gathering fault simulation and system statistics, since all data relevant to a particular node passes through it.

The task of the memory manager is to provide for several system services, like allocating and freeing chunks of memory. The memory manager is also used to obtain memory on other nodes when a node has run out of space in its own local memory. (see also chapter 4.4).

Both on-node and off-node memory allocation requests are handled equally by the memory manager. They share the same resource, the memory, and use the same allocation/deallocation algorithms. In order to reduce the number of messages in the fault simulator, each malloc request specifies the number of read requests (life span) for the particular fault list, which for purely combinational logic, is known. This enables the memory manager to free the allocated memory block automatically after receiving the right

number of read requests, without requiring an extra free message per fault list. No attempts are made for garbage collection in the event of an unsuccessful malloc request since, in most cases, the performance penalty would not justify the memory gain. A different approach is followed to insure better global memory utilization (chap 4.4).



The task of the router is to route message between different nodes. In our application, messages can have varying sizes. Most of them are short, 10 - 100 bytes, but some of them can be very long. Table 2 shows some message statistics. These statistics were collected on a 32 node partition of V256, by fault simulating five random patterns on one of the sample designs (C7522). With this many nodes, fault simulating one pattern takes about 0.7 seconds during which 1155 jobs are executed and about 1200 faults out of 7550 are caught. The columns show the average size of message, the range when the size is variable, the frequency with which the messages are sent and the type. The latter can be n to n (node to node),

h to n (host to node) or n to n/h (node to node or host). Each job requires the sending of many messages; on average 2.4 request messages per job and 2.3 result messages per job.

Messages can be sent from any node to any other node. Each message is sent as a unit and consists of a destination, a length and the actual message. The application program is responsible for providing the destination of the message but is not involved in actually getting the message to its destination. It merely sends the message to its on-node router. The router processes on the nodes then cooperatively route the message to its proper destination.

Message Description	Average size (bytes)	Range min-max (bytes)	Freq. per job	Freq. per pattern	Туре
input pattern	41.9	28 - 72		32	h-n
fault list request	20		2.4	2724	n-n
fault list	63.0	20-3280	2.4	2724	n-n
result message	24		2.3	2642	n-n/h

Table 2: Message characteristics

The main characteristics of the router are that messages are sent along a shortest path from the sender to the destination and that the routing is done in a strict store-and-forward fashion. The router reads the destination and determines whether the message has to be forwarded or sent to the application program. If the message has to be forwarded, the router determines the next neighbor to which to send the message by consulting a router table. The contents of the router tables depend on the node on which the router process resides and on the destination. They do not depend on the sender's address. The message is then forwarded as one unit and stored on the neighbor where it will be processed in a similar fashion by its router.

In addition, the routing is done in a deadlock-free fashion. The paths that a message can take are implicitly stored in the router tables and are restricted to make the routing deadlock-free. The deadlock avoidance algorithm that we implemented is the 2-plane scheme described by Yantchev et al. [22].

4. Simulation Software

4.1 Preprocessing

We parallelize fault simulation by assigning gates in the design to the nodes in the parallel processor. In fact, we improve the performance by combining gates in small single output clusters and assigning clusters rather than individual gates to the processors. The job associated with a cluster is the calculation of the output fault table, i.e. the table of faults that produce fault-effects on the output of the cluster. The input to a job are the logic values on the inputs of the cluster, the logic description of the cluster, faults on the input or output pins of the cluster and internal to the cluster and the fault tables for the input nets to the cluster.

How the assignment of clusters to processors is done strongly influences the performance of the simulator. Presently, the allocation is done randomly. This assignment of clusters is clearly not very good from a communication point of view: the average distance messages have to travel is of the order \sqrt{n} , where n is the number of nodes in the parallel machine. Better assignment algorithms are possible, but they have not been implemented yet.

4.2 Overall Simulation Flow

The simulation of the clusters that are assigned to a processor is controlled by the controller process CTL, shown in figure 1. When the design is loaded into the parallel processor, the controller process on each node builds several data structures and initializes them. After initializing the data structures, the controller blocks, waiting for an input pattern message or result message. These messages indicate that the fault table for some line that is an input to a cluster on that node has been evaluated. The message gives also the processor number where the fault list is located. These messages cause an update to take place for all clusters on that node to which the line fans out. For each cluster a counter keeps track of the number of input stems that have been evaluated. When all its inputs have been evaluated, a cluster is put in a FIFO buffer to be simulated. All clusters on that queue will be simulated locally except when dynamic load balancing modifies this allocation (chapter 4.5).

Clusters are taken from the queue and sent to the actual simulator, FSIM in the figure. When FSIM finishes its simulation, it sends a result message to the controller, which then updates its data structures. The controller also forwards the result message to those nodes that own clusters to which the output of the simulated cluster fans out. These messages will cause further updates and the simulation proceeds as above. The simulation of a pattern terminates when there are no more clusters on any ready queue and no more processors are working.

Notice that this method guarantees that the clusters are processed in topological order even when they were not ordered so in the controller's data structures. Once the queue contains some clusters, the mechanism of sending jobs, receiving result messages, putting clusters on the queue and taking them off is all that is required to keep the simulation going. This way, the fault simulation is cast into a Dataflow process, with the result messages functioning as tokens.

To make the controller as efficient as possible, the data-structures it works on are tailor-made for emulating the Dataflow process. All job descriptions are pre-stored in the local memory of the controller and filled in as much as possible when the design description is received by the controller. Once a cluster is ready to be processed, the job description can be sent to FSIM without any further alterations.

4.3 Fault Simulation

When the fault simulation process FSIM receives a job message, it parses it and then sends out requests for the input fault tables that it needs. While waiting for the input tables to return, the fault simulation process simulates the internal cluster faults. The faults in the input fault tables are processed in ascending order of fault numbers, so that a sorted fault table is produced at the end. The internal faults that were caught are merged into the final list, and the resulting fault table is stored at the node and a result message is sent to the controller process.

4.4 Global Memory Utilization

As mentioned previously, a good parallelization strategy should properly use the total available memory in the distributed system. This is a trivial requirement to fulfill in the cases where memory utilization patterns are "well-behaved", that is completely known at compile-time. This is typically the case in regular problems such as matrix manipulations, image processing and finite element method analysis. The problem becomes difficult in fault simulation since the amount of memory required cannot be determined a-priori, and fluctuates widely from one test pattern to another. It is possible for the memory of one processor to overflow with malloc requests, while the memories of other processors are under-utilized. Such nondeterministic behavior could cause a particular run of the fault simulation to abort while, in a global sense, there is enough memory to accommodate the simulation. Memory compaction techniques would temporarily alleviate the problem, at the expense of performance, but would not solve it.

In order to achieve good global utilization of the available memory, the following memory overflow suppression (MOS) approach is taken. Upon an unsuccessful on-node malloc request, the MMU issues an off-node request. The node receiving the off-node malloc services it and returns an acknowledge message when the malloc is successful. If the node did not have enough memory, it forwards the original request to another processor. The scheme proceeds until a success is reported back to the originator of the request, which then sends its fault table to the node where the malloc succeeded.

The efficiency of this method rests on the algorithm used in determining the processor to

send/forward the off-node malloc to. In the current implementation, the next processor is determined randomly and the search for an acceptor is terminated after a pre-fixed number of trials.

4.5 Load Balancing

Static assignment of clusters to nodes runs the risk of severe load imbalance. When a processor has no clusters on its ready queue but has not yet processed all its clusters, it should ask other processors, according to some protocol, for work. This protocol can be the same as the one used to find off-node storage. If a processor receives such a request, and if it has enough clusters on its ready queue, it sends a job message to the requestor. Any job message is such that the FSIM process that handles it does not need to know to which node this job was originally allocated. When it finished its job, it sends a result message back to its controller. Only this controller knows whether the job was originally assigned to it node or not. In the latter case it forwards the result message to the controller on the node where the job originated. The latter controller then handles the result message in the usual way.

5. Experimental Results

The parallel fault simulation program has been exercised on various logic designs. We will discuss here the results for the two largest ones. The circuit characteristics for these two designs are given in table 3. C7522 is the largest design in the ISCAS suite of test generation benchmarks [23]. DESIGNA is an internal design and is almost four times larger than C7522.

	C7522	DESIGNA
# clusters	1155	5454
average cluster size	3.8	2.7
# faults	7550	26299
Table 3: Desig	n statistic	s

5.1 Overall performance

Table 4 shows the simulation time per pattern, measured by simulating five random input patterns and taking their average simulation time. The simulation time is measured from the moment an input pattern is send to V256 to the moment the result

message is returned. Clearly, the single pattern simulation time for DESIGNA is not four times as much as it is for C7522. To understand this improved performance and also to understand the actual speedup, more detailed statistics have to be taken.

Three different times were measured. First of all, the time to complete a single job, i.e. the simulation of one cluster including the requesting and receiving of required input fault tables. These times are most conveniently measured in the CTL process: it is the time lapse between the sending of a job description to FSIM and the receiving of the corresponding result message. Secondly, we want to know how much time is spent idling, i.e. waiting for another job to become ready. Such idling occurs when a processor still has some clusters to process but all of them need input tables that have not been computed yet. These idle times are again most conveniently measured by CTL.

Finally, we want to know how much time FSIM spends waiting between sending requests for externally stored fault tables and receiving them. These wait times are measured by FSIM itself. In fact, FSIM measures two distinct waiting times. The first one is the total lapse between sending out the messages and receiving the replies. FSIM does some

work after requests have been sent out and the time spent doing that work should not be counted as wait time. The second lapse time is the real wait time, i.e. the time between finishing this additional work and receiving the fault tables.

Results for both designs are shown in table 4. Clearly, the main difference between the two designs is in the average job times and the number of idle periods. The difference in average job time results from the smaller average size of the clusters in DESIGNA: the average time per gate is roughly the same in both designs. More importantly, DESIGNA has relatively fewer idle periods than C7522. This is to be expected, because a large number of idle periods indicates a lack of parallelism, which in larger designs is less likely than in smaller ones. In fact, in C7522 the idle periods account for about 25 % of the total lapse time, while in DESIGNA they account for only 12 %.

Both idle times and wait times indicate load imbalance. For larger designs the idle times and the number of idle periods will decrease and therefore are of no real concern. The influence of the waiting times will be discussed in the next section.

		C7522	DESIGNA
Total simulation time (secs.)		0.7	2.0
Average job time (msecs.)	:	9.50	7.65
Total number of idle per.	:	271.2	404
Average idle period (msec.)	:	21.57	18.35
Maximum idle period (msec.)			202.94
Average request time (msec.)		5.83	5.42
Maximum request time (msec.)	:	45.95	53.63
Average wait time (msecs.)	:	3.44	3.47
Maximum wait time (msecs.)	:	40.32	46.72

5.2 Performance analysis

The wait periods depend on how much time it takes for fault list requests to reach their destinations and for the requested fault lists to travel to the requesting node. This is a fundamental problem. The minimum time any simulator needs is governed by the longest paths in the design. As no node on any such a longest path can be simulated before the previous node on that path has been simulated, no

parallelization can reduce the time needed the simulate the nodes on such a path. With the random assignment of gates to processors employed here request times are proportional to d(P), the average distance in a mesh of P nodes, and the minimum time needed to do the fault simulation will therefore nave a term proportional to d(P) as well.

The total time T taken by the fault simulation depends on the number of processors P, the number of clusters C and the allocation of the clusters to the

processors. In practice, C is roughly one third of the number of gates in the design. The time needed for the simulation of one cluster is roughly the sum of the time needed to obtain the fault tables from other nodes and the time needed for processing these fault tables. When P becomes large, the first term will dominate and we will focus on its effects.

We therefore find:

$$T \propto \frac{d(P)}{P} CS,$$
 (1)

where we assumed that there is perfect load balance. The total time taken on one processor is also proportional to C and S, and the resulting speedup is therefore proportional to P/d(P). With the random allocation on a rectangular mesh, d(P) is roughly equal to $2\sqrt{P}/3$, and the speedup is proportional to \sqrt{P} .

The most interesting application of parallel processors to fault simulation occurs when we let P grow linearly with C. This a very natural thing to do, because, when C increases, the amount of memory required to hold the design description has to increase as well. This is true even for uni-processors. In a parallel processor, a node typically has a fixed amount of memory and the easiest way to increase the total memory is therefore to increase the number of nodes P. When P is proportional to C, the total time taken for the fault simulation grows only as $SC^{0.5}$ rather than as SC when done on a uni-processor. Note also that on topologies with $d(P) \approx \ln P$ the total simulation grows only as $S \ln C$.

5.3 Speedup

Figure 2 shows the speedup as function of the number of processors. The speedup is calculated as follows. First, the total simulation time as seen from the host is obtained. The speedup is then measured by dividing the total simulation time at some fixed number of processors by the simulation time at the actual number of processors. Because these designs are too large to run on a single processor, the speedup with respect to the single node parallel processor could not be measured. Instead, the speedup is calculated relative to 32 nodes and the speedup at 32 nodes is set to 8. This arbitrary speedup was obtained by calculating P/d(P) (see the section on the performance analysis) and using for d(P) the value for random allocation on a rectangular mesh (=4).

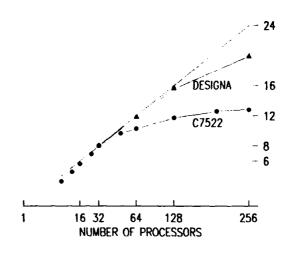


Figure 2: Relative performance

Speedup data are plotted in figure 2 as a function of P/d(P). The figure shows that for a range of P values, the speedup behaves roughly as P/d(P), as was found in the previous analysis. For small P, the speedup is not proportional to P/d(P) because the analysis was only correct for large P. For very large P, perfect load balancing cannot be maintained, because of the finite degree of parallelism in the design. However, as shown by the figure, larger designs keep their parallelism longer than smaller ones.

5.4 Scaled speedup

Finally, we would like to measure the actual speedup that is feasible with this parallelization. The maximum speedup shown when all 256 nodes are used is not realistic because of the severe underutilization of most of the nodes. We therefore consider the simulation times at the number of processors where the speedup curve starts to flatten out (32 for C7522 and 128 for DESIGNA). We could not run these designs on a single node, but by subtracting the wait times from the average job times and then multiplying the result by the number of jobs, we can estimate how long the fault simulation would take on a uni-processor. For C7522 we find about 7 seconds and for DESIGNA about 22.8 seconds. This leads to an estimated real speedup of 10 for C7522 and 22 for DESIGNA.

Acknowledgements

We would like to thank Gail Irwin, Dennis Shea, Winfried Wilcke and Deborra Zukowski for their help and support.

Bibliography

- [1] Alexander Miczo, Digital Logic Testing and Simulation Harper & Row, chap. 4.8, 1986.
- [2] Zcev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joseph D. Rutledge, "IISS A High-Speed Simulator," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 601-617, July 1987.
- [3] John A. Waicukauski, Edward B. Eichelberger, Donato O. Forlenza, Eric Lindbloom, and Thomas McCarthy, "Fault Simulation for Structured VLSI," VLSI Systems Design, pp. 20-32, December 1985.
- [4] Nagisa Ishiura, Masaki Ito, and Shuzo Yajima, "High-Speed Fault Simulation Using a Vector Processor Design," *Proceedings International Conference on Computer Aided*, pp. 10-13, IEEE, November 1987.
- [5] Raja Daoud and Fusun Ozguner, "Highly Vectorizable Fault Simulation on the Cray X-MP Supercomputer," *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 1362-1365, December 1989.
- [6] F. Ozguner, C. Aykanat, and O. Khalid, "Logic Fault Simulation on a Vector Hypercube Multiprocessor," Proceedings 3rd. Conference on Hypercube Concurrent Computers and Applications, pp. 1108-1116, ACM, January 1988.
- [7] Prathima Agrawal, Vishwani D. Agrawal, Kwang-Ting Cheng, and Raffi Tutundjian, "Fault Simulation in a Pipelined Multiprocessor System," Proceedings International Test Conference, pp. 727-734, IEEE, August 1989.
- [8] D. L. Ostapko and Z. Barzilai, "Fast Fault Simulation in a Parallel Processing Environment," *Proceedings International Test Conference*, pp. 293-298, IEEE, September 1989.
- [9] Patrick A. Duba, Rabindra K. Roy, Jacob A. Abraham, and William A. Rogers, "Fault Simulation in a Distributed Environment," *Proceedings 25TH Design Automation Conference*, pp. 686-691, ACM/IEEE, 1988.

- [10] Srinivas Patil and Prith Banerjee, "Fault Partitioning Issues in and Integrated Parallel Test Generation/Fault Simulation Environment," Proceedings International Test Conference, pp. 718-726, IEEE, August 1989
- [11] R. B. Mueller-Thuns, D. G. Saab, R. F. Damiano, and J. A. Abraham, "Portable Parallel Logic and Fault Simulation," *Proceedings ICCAD*, pp. 506-509, IEEE, November 1989.
- [12] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms* Addison-Wesley, chap. 6.6, 1983.
- [13] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," Proceedings 14th Design Automation Conference, pp. 462-468, IEEE, 1977.
- [14] Alexander Miczo, Digital Logic Testing and Simulation Harper & Row, chap. 4.8.3, 1986.
- [15] W. W. Wilcke, D. G. Shea, R. C. Booth, D. H. Brown, M. E. Giampapa, L. Huisman, G. R. Irwin, E. Ma, T. T. Murakami, I. Nair, F. T. Tong, P. R. Varker, and D. J. Zukowski, "The IBM Victor Multiprocessor Project," *Proceedings Fourth Hypercube Conference*, 1988.
- [16] INMOS Limited, Transputer Reference Manual Prentice Hall, 1988.
- [17] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2* Adam Hilger, chap. 3.5.5, 1988.
- [18] INMOS Limited, Transputer Reference Manual Prentice Hall, chap. 3, 1988.
- [19] INMOS Limited, OCCAM 2 Reference Manual Prentice Hall, 1989.
- [20] Dick Pountain and David May, A tutorial introduction to OCCAM programming INMOS, 1988.
- [21] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, pp. 666-677, August 1978.
- [22] Jelio Yantchev and Chris Jesshope, "Adaptive, Low Latency, deadlock-free Packet Routing for Networks of processors," < Missing journal>, < Missing year>.
- [23] Franc Brglez, Philip Pownall, and Robert Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *Proceedings of ISCAS*, pp. 695-698, IEEE, 1985.

Determination of Algorithm Parallelism in NP-Complete Problems For Distributed Architectures

R. Andrew Beard Gary B. Lamont

Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

Abstract

This paper explores the methods used to parallelize NP-complete problems and the degree of improvement that can be realized using a distributed parallel processor (hypercube) to solve these combinatoric problems. Common characteristics of NP-complete problems are identified and the set covering problem (SCP) is chosen as the vehicle for exploration. The SCP has application in many AI, communcations, computer science, and control problems and has been extensively studied in the serial case but a parallel implementation has not been reported. The design process states the basic algorithms in terms of UNITY metaprograms and iteratively develops three increasingly complex parallel versions of the SCP: coarse grain/static allocation, fine grain/dynamic allocation, and a dynamic load balancing version. A speedup is obtained in each of five test inputs with super-linear speedup obtained in four of the five tests.

Introduction

This paper explores the methods used to parallelize NP-complete problems and the degree of improvement that can be realized using a distributed parallel processor to solve these combinatoric problems. Many problems in AI, communications, computer science, control, and VLSI involve problems that reflect, in the worst case, an enumeration of all possible paths to a solution; that is, a combinatoric explosion whose associated solution time characteristics is bounded by an exponential function. General examples include: the set covering problem (SCP), the assignment prob-

lem, and the traveling salesman problem (TSP). Serial solutions to these problems are well known and documented for specific cases as well as for the general case [2, 5, 8, 13]. Specific parallel implementations of the assignment problem and the traveling salesperson problem have been reported [7, 10, 20, 11].

In the following sections, a brief background is presented followed by a discussion of the parallel solution techniques. The SCP is explained and three parallel SCP algorithms are discussed. The final sections present the performance of the parallel SCP programs and the utility of the parallel programs.

NP-Complete Problems

"It is an unexplained phenomenon that for many of the problems we know and study, the best algorithms for their solution have computing times which cluster into two groups" [15]. The solution time for the first group of problems is bounded by a polynomial-time function. For example, sorting — $O(n \log n)$, binary searching — $O(\log n)$, and matrix multiplication — $O(n^{\log 7})$. The second group of problems are those whose best known algorithms are nonpolynomial. For example, the TSP — $O(n^2 2^n)$, 0/1 knapsack problem — $O(2^{\sqrt{n}})$, and the SCP — $O(2^n)$ [15]. The thrust of this paper is a collection of problems in the second class termed nondeterministic polynomially-complete (NP-complete).

All NP-complete problems have two distinguishing characteristics. First, an NP-complete problem must be in the class \mathcal{NP} . Secondly, any NP-complete problem must be transformable to all other NP-complete problems in polynomial time and vice-versa [2].

Many NP-complete problems exhibit common characteristics which can be exploited or have an impact on a parallel implementation. Such characteristics include polynomial-time a priori reductions which potentially reduce the input problem, graph search, collection and use of partial state information (deterministic/estimation), and the unpredictable nature in which an NP-complete search progresses. In many instances, the a priori reductions are matrix manipulation operations and experience by other authors [6, 12, 16, 21] has shown that numerous matrix operations can be parallelized. Hence, it is reasonable to assume that the a priori reductions can be efficiently implemented on a parallel computer. NP-complete search methods, in general, utilize partial state information in conjunction with a bounding function or lower bound test to improve the efficiency of the search. The availability of selected partial state information obtained in other processors could potentially increase the efficiency of such bounding functions. Finally, the unpredictable nature of the search makes a coarse grain data partitioning algorithm inefficient for most problems; therefore, some method of dynamic load balancing is usually necessary to distribute sections of the search tree to idle processors.

To study the parallelization of NP-complete problems requires the selection of a representative problem which is proven NP-complete. The SCP was chosen for this research because many applications such as graph coloring, information retrieval, optimal resource scheduling, AI, circuit simulation, operations research, assignment problems, and VLSI logic expression simplification can be structured as an SCP problem. In addition, its generic NP-complete common characteristics are well documented [8] and a parallel implementation has not been reported.

Solution Techniques

Parallel programming design techniques involve decomposing the problem and developing the parallel algorithms. The major components of a parallel solution are developed in a four phase process. In the first phase, a meta-level design is accomplished using an appropriate design language such as UNITY (Unbounded Nondeterministic Iterative Transformations). UNITY is a design syntax developed by Chandy and Misra [6] for use in developing parallel programs. It is their attempt to incorporate a formal syntax into the parallel program design process and is similar, in many respects, to the Hoare's [14] method of designing concurrent sequential processes. Both methods are based on predicate calculus, temporal logic, and structured

design methods.

The second and third phases of the design iteratively transform the UNITY metaprograms into more complex UNITY representations of the problem until the UNITY metaprograms are sufficiently developed to map directly to a target architecture. The algorithms for this research are implemented on an Intel iPSC/2 hypercube; hence, the UNITY design is mapped to a cube-connected architecture. The final design phase is to conduct a complexity analysis of the algorithms.

Preprocessing

In many instances, the efficiency of the search techniques may be improved through the use of precomputation or preconditioning in the form of a priori reductions and selective bounding functions [5].

In many problems, it is possible to reduce the amount of searching required with problem specific reduction techniques or precomputation to reduce the dimensions of the original graph or tree [5]. One such reduction is to remove any states which are included in every branch of the search tree. For example, consider a tree search in which every branch contains the same node, say node 1. Since node 1 is contained in every solution to the search, it is not necessary to include this node in every search path. Rather, the node is removed from from the input problem and retained for later insertion into the final solution.

Dominance testing is a precomputation method which may decrease the size of the search tree by comparing the current state of the search to previously saved states. For instance, if the current state is a subset of a previous state and the current state's cost is greater than or equal to the previous state's, then the algorithm can backtrack. This technique requires a list of previous states be maintained in some suitably arranged manner (list) to allow an efficient comparison to the current state. If desired, all previous states may be saved; in which case, this approach resembles a breadth-first search of the problem space. As with most engineering problems, some tradeoff must be made between the number of stored previous states and the computation time required to check for dominance [9].

In addition to dominance testing, the computation of a lower bound is sometimes useful in bounding the search. A lower bound is the lowest possible cost down a branch of the search tree. Whether or not the lower bound can actually be obtained is irrelevant. What does matter is that if the current cost plus the lower bound exceeds the best cost obtained thus far, the algorithm backtracks. As the computation of the lower bound becomes more accurate, more branches of the

search tree are pruned. In the best case, the lower bound is exact and the search proceeds down the optimum path without backtracking. It is natural to assume that the precision of the lower bound computation is inversely proportional to the amount of computation time required to compute the lower bound. That is, a precise lower bound may require a long time to compute. Therefore, a suitable lower bound computation is one in which the time required to compute the bound does not adversely impact the overall search time [5, 9].

Load Balancing

As previously stated, NP-complete problems are inhomogeneous; therefore, it is difficult to balance the workload between autonomous processors. The incorporation of a global best cost which is known by all searching processors further increases the likelihood of a load imbalance as noted by Lai and Sahni [17]. Hence, load balancing is an integral component in any parallel implementation of an NP-complete search. The load balancing may take the form of static or dynamic allocation of subgraphs or it may be in the form of a dynamic load balancing scheme. In either instance, the necessity to balance the load between processors is well documented [24, 20, 11, 19, 23, 18].

The Set Covering Problem

The set covering problem (SCP) is one of a large class of NP-complete problems [2] extensively studied in the late 1960's and early 1970's in connection with operational research problems such as airline and assembly line scheduling, design of computer systems, crew scheduling, and political districting are all types of problems which can be formulated as an SCP [9, 24, 3]. The SCP is the problem of finding the minimum number of columns in a 0-1 matrix such that all rows of the matrix are covered by at least one element from any column and the cost associated with the covering columns is optimal (minimum or maximum) [8]. A 0-1 matrix is a rectangular matrix in which a covered row is denoted by a '1' in the covering columns. If the rows in the matrix represent the vertices of a graph, the existence of an arc between any two vertices is denoted by a '1' in the column of the matrix. A worst case search requires that all combinations of the various sets be check. This number of combinations is the power set or 2ⁿ. As an example, Figure 1 shows a 0-1 matrix in which the rows are covered by several different combinations of columns. The worst case search would be $O(2^8)$. Columns 0, 1, 2, 3, and 4 form a cover with a total cost of 27. The

optimal cover is formed by columns 0, 3, and 4 with a cost of 15.

		Columns							
		0	1	2	3	4	5	6	7
	0	1	1	1	0	0	1	0	1
	1	1	0	1	0	0	1	0	1
Rows	2	0	0	0	1	0	0	0	0
	3	0	1	0	0	1	0	1	1
	4	0	0	0	0	1	1	1	0
	5	1	1	0	0	0	0	1	0
		4	7	5	8	3	2	6	5
		Costs							

Figure 1: 0-1 Matrix [8]

A branch-and-bound search for a set cover attempts to minimize the number of set combinations tested in the search tree. It could be argued that all optimal search techniques are elaborate bookkeeping exercises. The branch-and-bound algorithm must store the traversed states so they can be recalled during the backtracking phase. Furthermore, it is desirable to choose only those sets which actually contribute to the solution. For instance, in Figure 1, suppose the search algorithm has chosen sets $\{0, 1\}$ to cover rows $\{0, 1, 3, 5\}$. It is pointless to choose set $\{2\}$ since it will not cover any rows not already covered by sets $\{0, 1\}$. Therefore, the efficiency of the search process is improved if there exists some method to choose the next set that covers rows not already covered.

Christofides [8, 9] suggests the construction of a table to assist in the bookkeeping and selection of the next set. The construction of the SCP table essentially preorders the rows and columns of the input matrix which guides the search in an efficient manner. The result is a variation of a best-first search without the requirement to maintain a priority queue ("open list").

The algorithm to build the table defines a block for each row of the matrix. All columns covering a particular row are contained in the block for that row. A search algorithm which selects one column from each block is guaranteed to cover all the rows. If, in addition to just selecting columns from the blocks, the search algorithm keeps track of the rows already covered, the algorithm could skip blocks which correspond to rows already covered. The search progresses from left to right in the table continually selecting and marking one column from each block as necessary. If the algorithm must backtrack, it regresses from right to left until it has found a block that can be further expanded. Notice, also, that the columns within each block are ordered in ascending order on the cost. This ordering,

in most cases, decreases the number of expanded nodes in the search tree. The worst case, of coarse, requires that all columns be checked before the optimal solution is found. As stated, the purpose of the SCP table is to assist the search in the bookkeeping and selection of the next column.

The SCP may be defined as follows [8]:

Given a set $R = \{r_1, r_2, ..., r_m\}$ and a family $\mathcal{L} = \{S_1, S_2, ..., S_N\}$ of sets such that $S_j \subset R$, any subfamily of $\mathcal{L} = \{S_{j1}, S_{j2}, ..., S_{jk}\}$ such that

$$\bigcup_{i=1}^k S_{ji} = R \tag{1}$$

is called a set covering of R. Given a 0-1 matrix,

Minimize:
$$z = \sum_{j=1}^{N} c_j \nu_j$$

Subject to: $\sum_{j=1}^{N} t_{ij} \nu_j, i = 1, 2, ..., m$

That is, minimize the cost such that all the elements of R are covered by at least one set from \mathcal{L} .

Since the parallel SCP programs are derived in part from serial SCP programs, a serial SCP UNITY program is developed first and then transformed to a parallel UNITY program. The first parallel UNITY program is not specific to any architecture; hence, an additional iteration is performed to develop an architecture specific UNITY program. Since the SCP is implemented on an iPSC/2 cube-connected computer, the UNITY program is designed to take advantage of the distributed nature of a cube-connected computer. The UNITY programs are quite extensive and are not presented here; rather, they may be found in Beard [4]. Following the development of each UNITY program, an invariant, a fixed point, and a progress condition are derived and employed to prove the correctness of the UNITY program. One of the strengths of UNITY is the ability to build on previous proofs; hence, at each iteration, the new program is proven correct based on the proof of the previous program.

Parallel Algorithms

For the major SCP process components, the UNITY design and subsequent translation are mapped to appropriate algorithms. These components include two a priori reductions, a bitonic merge sort, and a multifaceted search technique. The a priori reductions

are divide-and-conquer algorithms using a logarithmic collection technique, the bitonic merge sort is a generic implementation of the algorithm presented by Quinn [21], and the parallel search for the optimal set cover is an extension of the branch-and-bound algorithm presented by Christofides [8]. The parallel search utilizes a dominance test, a lower bound test, and a global best cost maintained at a central location for distribution to all processors.

As with the development of the UNITY programs, a serial SCP algorithm is developed first followed by the development of three increasingly complex parallel algorithms. Each new version of the parallel SCP is based on the previous version and is aimed at reducing individual processor idle-time. The first parallel implementation of the SCP employs a coarse grain algorithm with static allocation of the search space. The second version is a fine grain algorithm with dynamic allocation of the search space, and the third version is a fine grain algorithm with the addition of a dynamic load balancing technique in which the searching processors nondeterministically share portions of the search tree.

All three parallel algorithms employ a common control structure. Processor 0 is reserved as a controller with the remaining processors executing only the search algorithm. The controller receives the input matrix from the host processor and, following any user requested reductions, it coordinates a parallel bitonic merge sort of the rows and columns and sends the data to the searching processors. Depending on the particular algorithm, as discussed below, it may or may not partition the input state space. In either case, it functions as the central repository for the globally maintained best cost and the corresponding list of covering sets. As the searching processors search their respective search trees, they compare their local best cost against their copy of the global best cost. If a searcher finds a better solution than its copy of the current global best solution, it submits the solution to the controller. The controller compares all received costs against its current global cost and retains the better. If a new global best cost is received, this cost is broadcast to all searching processors for use in bounding their search trees. The following sections describe the three parallel algorithms.

Coarse Grain/Static Allocation

The coarse grain algorithm is the simplest of the three. Once the searching processors receive the sorted input matrix, each processor builds a copy of the SCP table and expands the state space with the results of the expansion stored in a queue. The expansion algorithm is a simple breadth-first expansion of the state space which divides the search tree by first inserting

the level 1 nodes into a queue. If necessary, the expansion algorithm continues to expand the tree by removing the top entry from the queue and expanding it to the next level. The expansion is complete when the number of subtrees is greater than the number of searching processors or a preset number of subtrees exist.

As stated, the same expansion algorithm is executed by all searching processors; therefore, the queue on each searcher is identical. This duplication is not necessarily globally efficient but the algorithm is simple to develop and duplicate on all processors. Following the initial expansion of the search tree, the searching processors remove subtrees from the queue based on their processor ID. For example, given a queue with three entries and two searching processors, processor 1 removes the first item from the queue and processor 2 removes the second item from the queue. When finished with the current subtree, processor 1 removes the third subtree from the queue. Processor 2 sits idle following the completion of its search.

In this load balancing scheme, the initial search tree is divided a predetermined number of times and all searching processors may or may not receive a subtree to search. The subtrees are statically allocated to the searching processors since the allocation of subtrees to processors is determined in the algorithm (i.e., 'hard-coded').

As one might suspect, much time is wasted by idle processors and the workload is far from balanced for most problem instances. Therefore, the coarse grain algorithm is modified to decrease the processor idletime.

Fine Grain/Dynamic Allocation

The initial expansion for the fine grain algorithm is quite similar to the breadth-first expansion for the coarse grain algorithm. One major difference is worth noting. In an attempt to converge on the optimal solution quicker, the new expansion algorithm combines both a breadth-first and a depth-first expansion. Given the matrix preordering and the construction of the SCP table previously explained, it is likely that the optimal solution to the SCP lies in the left-most portion of the search tree. Therefore, at each level in the search tree, the expansion algorithm accomplishes a breadth-first expansion on the left-most node in the search tree.

The dynamic allocation portion of the search attempts to decrease overall processor idle-time. The expansion algorithm is moved from the searching processors to a controlling processor (supervisor). The subtrees are constructed and the controller assigns each processor a initial subtree. As processors finish,

they are assigned another subtree from the queue until all the queue is empty. At which point, increasing numbers of processors become idle during the wind-up phase of the search. Intuitively, this search algorithm should perform better than the coarse grain algorithm with static allocation since processor idle-time will decrease.

Dynamic Load Balancing

The dynamic load balancing version of the SCP begins as a fine grain parallel algorithm and enters a dynamic load balancing process when all subgraphs have been distributed. Upon completion of the fine grain distribution of subgraphs, the controller triggers the active participation of a separate process called the token process. The token process exists on all processors and its only function is to coordinate the dynamic load balancing scheme. A token is circulated (ring) through all nodes in the cube and is composed of a linear array of m integers where m is the number of processors in the user acquired cube. The first integer in the token (Token[0]) denotes the number of searchers still searching and is included to facilitate quick checking of the status of the search. The remaining integers in the array are used by the searchers to indicate whether they are working or idle.

The token process located on the controller (i.e., processor 0), initally examines the first element in the token to determine if any searchers are still searching. If all searchers are waiting for another subgraph (Token[0] = 0), the search is complete and the token process notifies all searching processors. If any searcher is still working (Token[0] \neq 0), the token is passed unchanged to the next node in the ring.

Each token process residing on the searching processors continually monitors its receive buffer for the presence of the token, a request from another processor, or a message stating that the current processor is idle. Should the token arrive and the processor is idle, the token is updated and the token process cycles down the linear array until it finds a processor still searching or it has polled all active processors.

If the token process finds a working processor, it requests a subgraph from that processor. The requesting token process communicates with the token process located on the active processor. The searching processes are never allowed to communicate with each other. All dynamic load balancing is coordinated through the token process. The result is an efficient, simple, and highly reusable dynamic load balancing scheme.

The token processes coordinate/control the dynamic load balancing process; however, the searching process is responsible for partitioning the search tree for sharing. Two problems must be addressed. First, the shar-

ing algorithm must ensure completion of the search. In other words, a race condition must not develop where the same subtree is continually passed between processors. Second, it is desirable to limit the amount of unnecessary sharing.

The race condition is easily prevented. A searching process is only allowed to share after it has backtracked. This requirement forces the searching process to expand at least one branch in the search tree and, since any shared subtree consists of unexpanded nodes, a race condition can not occur.

To limit the amount of sharing, the algorithm which partitions the subtrees is designed to partition the largest possible subtree. When a subtree is requested, an active searching process backtracks through the search tree until it finds the highest expandable node. The beginning of this new subtree is marked so that it will not be searched by the current processor and the (largest expandable) subtree is given to the token process which transmits it to the requesting token process.

Performance

A serial version and the three parallel algorithms for the SCP were executed on a 32-node Intel iPSC/2 computer. Since the clock on the host measures relative process time and the node processor's clock measures absolute time, on would expect the serial program execution time to be equivalent on both the node processor and the host. In most cases, the host search time was longer probably due to the context switching of other user processes. Therefore, in order to accurately measure the speedup, an optimized serial version of the SCP is executed on one of the node processors and all time associated with communications is removed from the search time. A comparison between the serial program time executing on a node processor and the parallel program time decreases the measured speedup, but is a more accurate reflection of the actual speedup since the effects of the hardware and operating system are essentially eliminated.

Test Problems

Twenty-four 0-1 test matrices are generated to validate the effectiveness of the SCP algorithms. Five of these matrices require in excess of two hours to solve by the serial algorithm and are used to measure the efficiency of the three parallel SCP algorithms. The density of the matrices is defined as the total number of 1's divided by the total number of matrix elements and the columns are all assigned unit cost since unit cost problems are usually more difficult to solve. Three of the five test cases (matrices #1, #2, and #3) are

 100×100 matrices with densities of 0.28, 0.27, and 0.26 respectively. Matrix #4 is 75×125 with a density of 0.25 and matrix #5 is 70×70 with a density of 0.08.

Performance Metrics

For the purpose of this paper, the performance metrics of primary concern are the search time and the total execution. The search time is used to compute the speedup and the total execution time is used to measure the maximum processor idle-time. Clearly, other metrics are of interest such as the number of expanded nodes, the time spent sorting and preprocessing the input data, the time expended by the dynamic load balancing programs, number of times the global best cost was updated, and the time when the global best cover was last updated. These metrics are not considered here but a discussion may be found in Beard [4].

Results

Much of the justification for implementing three different parallel versions of the SCP is based on reducing individual processor idle-time. The results indicate that, in general, the coarse grain algorithm incurred the most idle-time (85–1065 seconds) followed by the fine grain algorithm (152–144 seconds) and then the dynamic load balanced (DLB) algorithm (19–42 seconds); however, these numbers are deceiving. If one were to judge the parallel algorithms based solely on processor idle-time, the DLB algorithm is clearly the most efficient algorithm and the fine grain algorithm is usually better than the coarse grain algorithm. Such a conclusion is invalid.

Figures 2, 3, 4, 5, and 6 show the normalized speedups obtained for each of the five test matrices described above for 1-31 searching processors. The legend is displayed to the right of each graph and is defined as follows: L—linear speedup, cg—coarse grain algorithm with static allocation, fg—fine grain algorithm with dynamic allocation and lb—dynamic load balanced algorithm.

Interpretation of the data from three of the five test cases (Figures 2, 3, and 5) show that even though the dynamic load balancing is effective in balancing the load, the additional processing necessary to accomplish this may actually increase the overall search time such that the dynamic load balancing algorithm is slower than the fine grain algorithm. Furthermore, three of the test cases (Figures 4-6) indicate that the coarse grain expansion algorithm is more efficient than the fine grain expansion algorithm. Notice also that four of the five test cases (Figures 3-6) exhibited superlinear speedup with the search of matrix #4 shown in Figure 5 obtaining the largest speedup at 61!

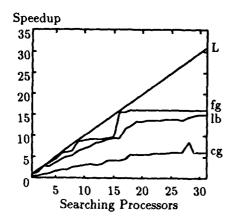


Figure 2: Matrix #1 Speedup

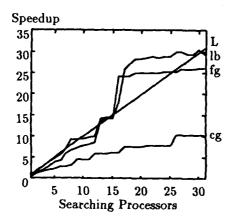


Figure 3: Matrix #2 Speedup

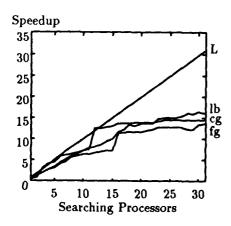


Figure 4: Matrix #3 Speedup

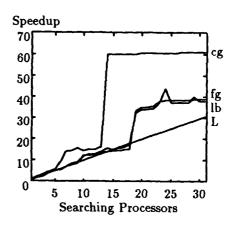


Figure 5: Matrix #4 Speedup

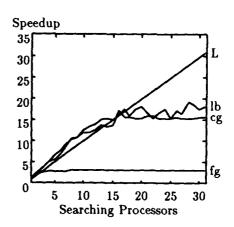


Figure 6: Matrix #5 Speedup

These seemingly erratic results are easily explained. Recall that the coarse grain and the fine grain versions of the SCP use different breadth-first expansion algorithms. Given that NP-complete problems are inhomogeneous, the different expansion algorithms produce radically different search graphs; hence, the difference in performance between the coarse grain and fine grain algorithms is unpredictable. In fact, one could argue that the two algorithms are searching entirely differently since the search graphs produced for the same problem are different.

The expansion algorithm affects both the solution time and the idle-time; however, the processor idle-time is affected more by the allocation of the initial subgraphs than by the expansion algorithm. Since subgraphs in the coarse grain algorithm are statically allocated, processor idle-times are typically longer with the coarse grain algorithm than with the dynamically allocated fine grain algorithm. Even so, an examination of the raw data reveals that the coarse grain expansion algorithm usually results in a quicker best cover time (i.e., it finds the optimal cover before the fine grain expansion algorithm).

The DLB algorithm was developed to further decrease maximum processor idle-time and to improve the efficiency of the search algorithm. However, Figure 2 shows that the fine grain algorithm is consistently faster than the DLB algorithm. Either the DLB is inefficient or the fine grain algorithm is highly efficient for this problem instance. In this particular problem, the fine grain expansion algorithm balances the load from the beginning of the search. Any additional load balancing (e.g., dynamic load balancing) simply steals CPU cycles from the search algorithm and delays the completion of the search.

The DLB algorithm is not necessarily inefficient; however, it does include additional code to dynamically share portions of a processor's search graph. Even though the timing data obtained from the node processors indicates an extremely small percentage of time devoted to the dynamic load balancing process, the data does not show the total processor time devoted to the token process. This time is significant in some problem instances as shown in Figures 2, 3, 4, and 5. In each of these graphs, the speedup of the fine grain algorithm closely parallels the speedup of the DLB algorithm. Furthermore, notice that the fine grain algorithm is frequently more efficient than the DLB algorithm even though the performance data from the searching processors indicates the DLB algorithm did in fact share subgraphs between searching processors. Two reasons for the DLB's apparent inefficiency are: 1) the token process is stealing too much time from the search process, 2) the searching processors are spending too much time partitioning and sending subgraphs to other processors. Unfortunately, the mclock() function does not provide a method to compute the CPU time consumed by the separate token process; hence, another method must be found to measure process time. The second reason suggests that the processors are partitioning the subgraph at too low a level in the search tree and a heuristic algorithm is required to prevent such low-level partitioning.

Despite the previous figures, Figure 6 shows that the DLB algorithm does work. For this specific problem, the fine grain expansion algorithm only creates 19 subgraphs due to limitations in the expansion algorithm. In effect, this is a coarse grain partitioning of the initial search graph. Since only 19 subgraphs are developed, the processors quickly become idle and the efficiency of the search suffers. With the DLB algorithm, the idle processors immediately receive a subgraph from the working processors and contribute to the search. Had the dynamic load balancing algorithm not been effective, the DLB's speedup curve would have paralleled the fine grain algorithm's curve as in previous graphs.

Summary of Results

The SCP has application in solving many 'realworld' and NP-complete problems. For example, airline and assembly line scheduling, design of computer systems, railroad-crew scheduling, and political districting are all types of problems which can be formulated as an SCP [9, 24, 3]. Furthermore, since the SCP is an NP-complete problem, it can be used to solve other NP-complete problems such as the assignment and graph coloring problems after proper transformation. The key to applying the SCP to any of these problems is to identify the items that must be covered by some subset of another list of items. Once the two lists of items are identified, they must be formulated as a 0-1 matrix with the items to be covered as the rows and the covering items as the columns. Additionally, the covering items must have some associated cost to identify their relative importance.

One of the objectives of this research was to investigate methods to parallelize NP-complete problems. Three methods are presented and a speedup is obtained for each. In fact, a super-linear speedup is obtained for four of the five test matrices. The possibility of super-linear speedup in branch-and-bound search problems was predicted by Lai and Sahni [17] but it is unclear whether anyone had confirmed this phenomenon via the test results from an actual implementation. This is not to say that the algorithms

developed for this research routinely produce a superlinear speedup. On the contrary, one could develop many test cases which would quickly disprove such a statement. However, the algorithms presented here show a tendency to go super-linear for input test cases that require a substantial amount of time to solve with a serial algorithm. More research is required to ascertain whether specific problem characteristics can be a priori exploited to obtain predictable super-linear speedup.

The performance increases presented here are the result of a different approach than that documented in much of the published literature [18, 22, 1, 19, 11]. The typical approach to parallelizing an NP-complete problem seems to center around the existence of a centrally maintained priority queue containing unsolved subbranches. The processors receive a subgraph, further partition the subgraph, and then transmit the newly partitioned subgraphs back to the centrally maintained queue. Such an approach is communications intensive as shown by Quinn [22]. The approach presented here is to partition the search space first and distribute the subgraphs to the individual processors. As such, the communications overhead becomes insignificant and the problem becomes compute bound. This simple but elegant approach to the initial load balancing is only possible because of the preordering (i.e., the construction of the SCP table) accomplished before the search. The result is a simple and highly efficient initial distribution of the load for many problem instances. The possibility of a similar preordering in other NP-complete problems is left for future researchers.

To date, much of the research into parallel branchand-bound algorithms has focused on the traveling salesman problem. The research presented here contains the first known parallel implementation of the SCP. Given the general application of the SCP to many different problems and the results published in this document, applications based on a parallel SCP (e.g., weapon to target assignment, optimal resource scheduling, VLSI expression simplification, and information retrieval) could achieve considerable performance increases. Furthermore, the methods presented here show that it is possible to realize a performance increase using control and data structures centered around something other than a centrally maintained priority queue.

The results further indicate that the performance of a parallel NP-complete search is highly dependent on the method chosen to distribute or balance the load between the processors. The initial distribution of subgraphs accomplished by the parallel SCP algorithms, in many of the test cases, is sufficient to ensure a 'good'

load balancing. However, as Figure 6 indicates, a dynamic load balancing algorithm is necessary in those instances where the initial distribution fails to obtain the desired load balance. The dynamic load balancing algorithm developed for this research is a much simpler algorithm than those presented by Felten [11] or Ma [18]. The algorithm employs a separate process to pass a token between the processors and to coordinate all load balancing. The separate token process is designed such that termination is easily detected and, in the absence of any other load balancing scheme, the DLB algorithm may provide acceptable performance.

The concepts used to develop the data and control structures for this design lend themselves very efficiently to solving general NP-complete problems in an effective manner. These concepts include the division of data and control for the search algorithms, as well as the load balancing algorithms required to achieve the most productivity from every node processor. For the a priori reductions, the data is partitioned out to the processors where a reduction is performed on the reduced problem. The results of the individual reductions are recombined in neighboring processors and further reduced. Each node search is essentially a serial algorithm searching a reduced section of the tree with knowledge of the lowest cost obtained by all processors. Finally, the inhomogeneous nature of NPcomplete problems forces the development of a load balancing algorithm. Many such algorithms are possible; however, the designer must balance the amount of time required to load balance against the time required to complete the search. The dynamic load balancing algorithm developed for this SCP research is simple, reusable, and effective.

We would like to thank the following Intel representatives: Tony Anderson, Ray Asbury, Sean Griffin, and Randy Hufford. Without their combined help and patience, the results presented here would not have been possible.

References

- [1] Abdelrahman, Tarek S. and Trevor N. Mudge. Parallel branch and bound algorithms on hypercube multiprocessors. In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 1492-1499. The Association for Computing Machinery, 1988.
- [2] Aho, Alfred B., John E. Hopcroft and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Massachusetts, 1974.

- [3] Balas, Egon and Manfred W. Padberg. On the set-covering problem. Operations Research, 20:1152-1162, 1972.
- [4] Beard, Ralph A. Determination of algorithm parallelism in NP complete problems for distributed architectures. Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, March 1990.
- [5] Brassard, Giles and Paul Bratley. Algorithmics: Theory and Practice. Prentice Hall, Englewood Cliffs, New Jersey, First edition, 1988.
- [6] Chandy, K. Mani and Jayadev Misra. Parallel Program Design: A Foundation. Addison-Wesley, Reading, Massachusetts, 1988.
- [7] Chen, Woei-Kae and Edward F. Gehringer. A graph-oriented mapping strategy for a hypercube. In The Third Conference on Hypercube Concurrent Computers and Applications: Volume 1, pages 200-209. The Association for Computing Machinery, 1988.
- [8] Christofides, Nicos. Graph Theory: An Algorithmic Approach. Academic Press, London, England, 1975.
- [9] Christofides, Nicos and S. Korman. A computational survey of methods for the set covering problem. Management Science, 21(5):591-599, January 1975.
- [10] Ercal, F., J. Ramanujam and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. In The Third Conference on Hypercube Concurrent Computers and Applications: Volume 1, pages 210-221. The Association for Computing Machinery, 1988.
- [11] Felten, Edward W. Best-first branch-and-bound on a hypercube. In The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2, pages 1500-1504. The Association for Computing Machinery, 1988.
- [12] Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. Solving Problems on Concurrent Processors. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
- [13] Garey, Michael R. and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco, California, 1979.

- [14] Hoare, C. A. R. Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
- [15] Horowitz, Ellis and Sartaj Sahni. Fundamentals of Computer Algorithms. Computer Science Press, Rockville, Maryland, 1978.
- [16] Hwang, Kai and Faye A. Briggs. Computer Architecture and Parallel Processing. McGraw-Hill, New York, 1984.
- [17] Lai, T-H and S. Sahni. Anomalies in parallel branch-and-bound algorithms. CACM, 27(6):594-602, March 1984.
- [18] Ma, Richard P., Fu-Sheng Tsung, and Mae-Hwa Ma. A dynamic load balancer for a parallel branch and bound algorithm. In The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2, pages 1505-1513. The Association for Computing Machinery, 1988.
- [19] Pangas, Roy P. and Wooster, E. Daniels. Branchand-bound algorithms on a hypercube. In The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2, pages 1514– 1519. The Association for Computing Machinery, 1988.
- [20] Pettey, Chrisila and Michael R. Leuze. Parallel placement of parallel processes. In The Third Conference on Hypercube Concurrent Computers and Applications: Volume 1, pages 232-238. The Association for Computing Machinery, 1988.
- [21] Quinn, Michael J. Designing Efficient Algorithms for Parallel Computers. McGraw-Hill, New York, 1987.
- [22] Quinn, Michael J. Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer. *IEEE Transactions on Computers*, 39(3):384-387, March 1990.
- [23] Schwan, Karsten, John Gawkowski and Ben Blake. Process and workload migration for a parallel branch-and-bound algorithm on a hypercube multicomputer. In The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2, pages 1520-1530. The Association for Computing Machinery, 1988.
- [24] Wah, Benjamin W., Guo-jie Li and Chee Fen Yu. Multiprocessing of combinatorial search problems. *Computer*, 18(6):93-108, June 1985.

Concurrent Implementation Of Munkres Algorithm

T. D. Gottschalk California Institute of Technology Pasadena, CA 91125

April 7, 1990

1 Introduction

The so-called Assignment Problem is of considerable importance in a variety of applications, and can be stated as follows. Let

$$\mathcal{A} \equiv \{a_1, a_2, \dots, a_{N_A}\} \tag{1}$$

and

$$\mathcal{B} \equiv \{b_1, b_2, \dots, b_{N_B}\}\tag{2}$$

be two sets of items and let

$$d_{ij} \equiv d[a_i, b_j] \ge 0, \ a_i \in \mathcal{A}, \ b_j \in \mathcal{B}$$
 (3)

be a measure of the distance (dissimilarity) between individual items from the two lists. Taking $N_A \leq N_B$, the objective of the assignment problem is to find the particular mapping

$$i \mapsto \Pi(i), \ 1 \le i \le N_A, \ 1 \le \Pi(i) \le N_B$$
 (4)

$$i \neq j \Rightarrow \Pi(i) \neq \Pi(j)$$
 (5)

such that the total association score

$$S_{TOT} \equiv \sum_{i=1}^{N_A} d[i, \Pi(i)]$$
 (6)

is minimized over all permutations Π .

For $N_A \leq N_B$, the naive (exhaustive search) complexity of the assignment problem is $O[N_B!/(N_B-N_A)!]$. There are, however, a variety of exact solutions to the assignment problem with reduced complexity $O[N_A^2N_B]$, (Refs.[1-3]). Section 2 briefly describes one such method, Munkres Algorithm [2], and presents a particular sequential implementation. Performance of the algorithm is examined for the particularly nasty problem of associating lists of random points within the unit square. In Section 3, the algorithm is generalized for concurrent execution, and performance results for runs on the MarkIII hypercube are presented.

2 The Sequential Algorithm

The input to the assignment problem is the matrix $D \equiv \{d_{ij}\}$ of dissimilarities from Eq.(3). The first point to note is that the particular assignment which minimizes Eq.(6) is not altered if a fixed value is added to or subtracted from all entries in any row or column of the cost matrix D. Exploiting this fact, Munkres solution to the Assignment Problem can be divided into two parts

M1: Modifications of the distance matrix D by row/column subtractions, creating a (large) number of zero enties.

M2: With $\{R_Z(i)\}$ denoting the row indices of all zeros in column i, construction of a so-called *Minimal Representative Set*, meaning a distinct selection $R_Z(i)$ for each i, such that $i \neq j \Rightarrow R_Z(i) \neq R_Z(j)$.

The steps of Munkres algorithm generally follow those in the constructive proof of P. Hall's theorem on Minimal Representative Sets.

The preceding paragraph provides a hopelessly incomplete hint as to the number theoretic basis for Munkres Algorithm. The particular implementation of Munkres algorithm used in this work is as described in Chapter 14 of Ref.[3]. To be definite, take $N_A \leq N_B$, and let the columns of the distance matrix be associated with items from list A. The first step is to subtract the smallest item in each column from all entries in the column. The rest of the algorithm can be viewed as a search for special zero entries (starred zeros Z^*), and proceeds as follows:

Munkres Algorithm

Step 1: Setup

- 1. Find a zero Z in the distance matrix.
- 2. If there is no starred zero already in its row or column, star this zero.

3. Repeat steps 1.1, 1.2 until all zeros have been considered.

Step 2: Z* Count, Solution Assessment.

- 1. Cover every column containing a Z^* .
- Terminate the algorithm if all columns are covered. In this case, the locations of the Z* entries in the matrix provide the solution to the assignment problem.

Step 3: Main Zero Search

- 1. Find an uncovered Z in the distance matrix and prime it, $Z \mapsto Z'$. If no such zero exists, go to Step 5
- 2. If No Z* exists in the row of the Z', go to Step 4.
- 3. If a Z^* exists, cover this row and uncover the column of the Z^* . Return to Step 3.1 to find a new Z.

Step 4: Increment Set Of Starred Zeros

1. Construct the 'Alternating Sequence' of primed and starred zeros:

 Z_0 : Unpaired Z' from Step 3.2.

 Z_1 : The Z^* in the column of Z_0

 Z_{2N} : The Z' in the row of Z_{2N-1} , if such a zero exists.

 Z_{2N+1} : The Z^* in the column of Z_{2N} .

the sequence eventually terminates with an unpaired $Z' = Z_{2N}$ for some N.

- 2. Unstar each starred zero of the sequence.
- Star each primed zero of the sequence, thus increasing the number of starred zeros by one.
- 4. Erase all primes, uncover all columns and rows, and return to Step 2.

Step 5: New Zero Manufactures

- 1. Let h be the smallest uncovered entry in the (modified) distance matrix.
- 2. Add h to all covered rows.
- 3. Subtract h from all uncovered columns
- 4. Return to Step 3, without altering stars, primes or covers.

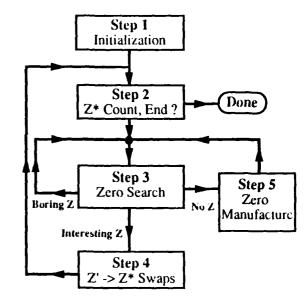


Figure 1: Flowchart for Munkres algorithm

A (very) schematic flowchart for the algorithm is shown in Fig.(1). Note that Steps 1,5 of the algorithm overwrite the original distance matrix.

The preceeding algorithm involves flags (starred or primed) associated with zero entries in the distance matrix, as well as 'Covered' tags associated with individual rows and columns. The implementation of the zero tagging is done by first noting that there is at most one Z^* or Z' in any row or column. The covers and zero tags of the algorithm are accordingly implemented using five simple arrays:

CC(k): Covered column tags, $1 \le k \le N_{COLS}$.

CR(j): Covered row tags, $1 \le j \le N_{ROWS}$

ZS(k): Z^* locators for columns of the matrix. If positive, ZS(k) is the row index of the Z^* in the k^{th} column of the matrix.

 $\mathbf{ZR}(j): Z^{\bullet}$ locators for rows of the matrix. If positive, $\mathbf{ZR}(j)$ is the column of the Z^{\bullet} in the j^{th} row of the matrix.

ZP(j): Z' locators for rows of the matrix. If positive, ZP(j) is the column of the Z' in the j^{th} row of the matrix.

Entries in the cover arrays CC and CR are one if the row or column is covered zero otherwise. Entries in the zero-locator arrays ZS, ZR and ZP are zero if no zero of the appropriate type exists in the indexed row or column.

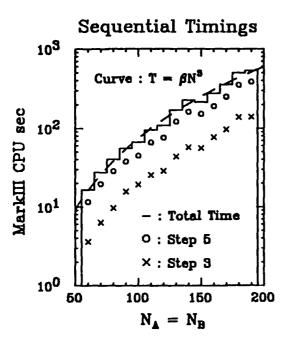


Figure 2: Timing results for the sequential algorithm versus problem size

With the Star-Prime-Cover scheme of the preceeding paragraph, a sequential implementation of Munkres algorithm is completely straightforward. At the beginning of Step 1, all cover and locator flags are set to zero, and the initial zero search provides an initial set of non-zero entries in ZS(). Step 2 sets appropriate entries in CC() to one and simply counts the covered columns. Steps 3 and 5 are trivially implemented in terms of the Cover/Zero arrays and the 'Alternating Sequence' for Step 4 is readily constructed from the contents of ZS(), ZR() and ZP().

As an initial exploration of Munkres algorithm, consider the task of associating two lists of random points within a 2D unit square, taking the cost function in Eq.(3) to be the usual Cartesian distance. Figure(2) plots total CPU times for execution of Munkres algorithm for equal size lists versus list size. The vertical axis gives CPU times in seconds for one node of the MarkIII hypercube. The circles and crosses show the time spent in Steps 5 and 3, respectively. These two steps (zero search and zero manufacture) account for essentially all of the CPU time. For the 190×190 case, the total CPU time spent in Step 2 was about 0.9 CPU sec, and that spent in Step 4 was too small to be reliably measured. The large amounts of time spent in Steps 3 and 5 arise from the very large numbers of times these parts of the algorithm are executed. The 190×190 case involves 6109 entries into Step 3 and 593 entries into Step 5.

Since the zero searching in Step 3 of the algorithm is required so often, the implementation of this step is done with some care. The search for zeros is done column-by-column, and the code maintains pointers to both the last column searched and the most recently uncovered column (Step 3.3) in order to reduce the time spent on subsequent re-entries to the Step 3 box of Fig.(1).

The dashed line if Fig.(2) indicates the nominal $\Delta T \propto N^3$ scaling predicted for Munkres algorithm. By and large, the timing results in Fig.(2) are consistent with this expected behavior. It should be noted, however, that both the nature of this scaling and the coefficient of N^3 are very dependent on the nature of the data sets. Consider, for example, two identical trivial lists

$$a_i \equiv b_i \equiv i, \ 1 \le i \le N \tag{7}$$

with the distance between items given by the absolute value function. For the data sets in Eq.(7), the preliminaries and Step 1 of Munkres algorithm completely solve the association in a time which scales as N^2 . In contrast, the random point association problem is a much greater challenge for the algorithm, as nominal pairings indicated by the initial nearest-neighbor searches of the malinal any step are tediously undone in the creation of the same rease-like sequence of zeros needed for Step 4 As brief, instructive illustration of nature of this processing, Fig.(3) plots the CPU time Per Step for the last passes through the outer loop of Fig.(1) for the 150×150 assignment problem (recall that each pass through the outer loop increases the Z^* count by one). The processing load per step is seen to be highly non-uniform.

3 The Concurrent Algorithm

The timing results from Fig.(2) clearly dictate the manner in which the calculations in Munkres algorithm should be distributes among the nodes of a hypercube for concurrent execution. The zero and minimum element searches for Steps 3 and 5 are the most time consuming and should be done concurrently. In contrast, the essentially bookkeeping tasks associated with Steps 2 and 4 require insignificant CPU time and are most naturally done in lockstep (i.e., all nodes of the hypercube perform the same calculations on the same data at the same time). The details of the concurrent algorithm are as follows.

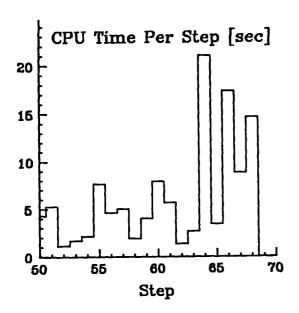


Figure 3: Times per loop (i.e., $N[Z^*]$ increment) for the last several loops in the solution of the 150×150 problem.

Data Decomposition

The distance matrix $\{d_{ij}\}$ is distributed across the nodes of the hypercube, with entire columns assigned to individual nodes. (This assumes, effectively, that $N_{COLS} \gg N_{NODES}$, which is always the case for assignment problems which are big enough to be 'interesting'.) The cover and zero locator lists defined in Section 2 are duplicated on all nodes.

Task Decomposition

The concurrent implementation of Step 5 is particularly trivial. Each node first finds its own minimum uncovered value, setting this value to some 'infinite' token if all columns assigned to the node are covered. A simple loop on communication channels determines the global minimum among the node-by-node minimum values, and each node then modifies the contents of its local portion of the distance matrix according to Steps(5.2,5.3).

The concurrent implementation of Step 3 is just slightly more awkward. On entry to Step 3, each node searches for zeros according to the rules of Section 2, and fills a 3-element status list:

$$L[j] \equiv L[Node_j] \equiv \{S, k_{ROW}, k_{COL}\}$$
 (8)

where S is a zero-search status flag,

$$S \equiv \begin{cases} -1 & \text{No } Z \text{ was found} \\ 0 & Z \text{ with } Z^* \text{ in row (Boring)} \\ 1 & Z \text{ without } Z^* \text{ (Interesting)} \end{cases}$$
 (9)

If the status is non-negative, the last two entries in the status list specify the location of the found zero. A simple channel loop is used to collect the individual status lists of each node into all nodes, and the action taken next by the program is as follows:

- If all nodes give negative status (no Z found), all nodes proceed to Step 5.
- If any node gives status 1, all nodes proceed to Step 4 for lockstep updates of the zero location lists, using the row-column indices of the node which gave status 1 as the starting point for Step 4.1. If more than one node returns status 1 (highly unlikely, in practice), only the first such node (lower node number) is used.
- If all zeros uncovered are 'Boring', the coverswitching in Step 3.3 of the algorithm is performed. This is done in lockstep, processing the Z's returned by the nodes in order of increasing node number. Note that the cover rearrangements performed for one node may well cover a Z returned by a node with higher node number. In such cases, the nominal Z returned by the later node is simply ignored.

It is worth emphasizing that only the actual searches for zero and minimum entries in Steps 3 and 5 are done concurrently. The updates of the cover and zero locator lists are done in unison.

The concurrent algorithm has been implemented on the MarkIII hypercube, and has been tested against random point association tasks for a variety of list sizes. Before examining results of these tests, however, it is worth noting that the concurrent implementation is not particularly dependent on the hypercube topology. The only communication-dependent parts of the algorithm are

- 1. Determination of the ensemble-wide minimum value for Step 5.
- 2. Collection of the local Step 3 status lists (Eq.(9).

either of which could be easily done for almost any MIMD architecture.

Table 1 presents performance results for the association of random lists of 200 points on the MarkIII

N[Nodes]	1	2	4	8
T[Total]	654.83	372.70	205.48	119.25
T[Step 3]	183.80	128.04	81.59	56.66
T[Step 5]	462.06	237.54	117.39	57.94
€Total	-	0.878	0.800	0.686
Step 3	-	0.718	0.563	0.405
^c Step 5	_	0.973	0.984	0.997
N[Step 3]	7075	4837	3483	2778

Table 1: Concurrent performance For 200×200 random points

hypercube for various cube dimensions. (For consistency, of course, the same input lists are used for all runs.) Time values are given in CPU seconds for the total execution time, as well as the time spent in Steps 3 and 5. Also given are the standard concurrent execution efficiencies,

$$\epsilon_N \equiv \frac{T[1 \text{ Node}]}{N \times T[N \text{ Nodes}]} \tag{10}$$

as well as the numbers of times the Step 3 box of Fig.(1) is entered during execution of the algorithm. The numbers of entries into the other boxes of Fig.(1) are independent of the hypercube dimension.

There is an aspect of the timing results in Table 1 which should be noted. Namely, essentially all inefficiencies of the concurrent algorithm are associated with Step 3 for 2 Nodes compared to Step 3 for 1 Node. The times spent in Step 5 are approximately halved for each increase in the dimension of the hypercube. However, the efficiencies associated with the zero searching in Step 3 are rather poorer, particularly for larger numbers of nodes.

At a simple, qualitative level, the inefficiencies associated with Step 3 are readily understood. Consider the task of finding a single zero located somewhere inside an $N \times N$ matrix. The mean sequential search time is

$$\langle T_{\text{Search}}[1 \text{ Node}] \rangle \propto (N \times N)/2$$
 (11)

since, on average, half of the entries of the matrix will be examined before the zero is found. Now consider the same zero search on two nodes. The node which has the half of the matrix containing the zero will find it in about half the time of Eq.(11). However, the other node will always search through all of its $N \times N/2$ items before returning a null status for Eq.(9). Since the node which found the zero must wait for the other node before the (lockstep) modifications of zero

N[Nodes]	1	2	4	8
T[Total]	68.08	38.79	23.11	16.40
T[Step 3]	19.63	13.09	9.69	8.00
T[Step 5]	44.99	22.99	11.79	6.16
€Total	-	0.878	0.736	0.519
€Step 3	-	0.750	0.506	0.307
^c Step 5	-	0.978	0.954	0.913
N[Step 3]	2029	1430	1134	991

Table 2: Concurrent performance For 100×100 random points

locators and cover tags, the node without the zero determines the actual time spent in Step 3, so that

$$\langle T_{\text{Search}}[2 \text{ Nodes}] \rangle \approx \langle T_{\text{Search}}[1 \text{ Node}] \rangle$$
 (12)

In the full program, the concurrent bottleneck is not as bad as Eq.(12) would imply. As noted above, the concurrent algorithm can process multiple 'Boring' Z's in a single pass through Step 3. The frequency of such multiple Z's per step can be estimated by noting the decreasing number of times Step 3 is entered with increasing hypercube dimension, as indicated in Table 1. Moreover, each node maintains a counter of the last column searched during Step 3. On subsequent re-entries, columns prior to this marked column are searched for zeros only if they have had their cover tag changed during the prior Step 3 processing. While each of these algorithm elements does diminish the problems associated with Eq.(12), the fact remains that the search for zero entries in the distributed distance matrix is the least efficient step in concurrent implementations of Munkres algorithm.

The results presented in Table 1 demonstrate that an efficient implementation of Munkres algorithm is certainly feasible. It is next interesting to examine how these efficiencies change as the problem size is varied.

The results shown in Tables 2,3 demonstrate an improvement of concurrent efficiencies with increasing problem size - the expected result. For the 100×100 problem on 8 nodes, the efficiency is only about 50problem is too small for 8 nodes, with only 12 or 13 columns of the distance matrix assigned to individual nodes.

While the performance results in Tables 1-3 are certainly acceptable, it is nonetheless interesting to investigate possible improvements of efficiency for the zero searches in Step 3. The obvious candidate for an algorithm modification is some sort of checkpoint-

N[Nodes]	1	2	4	8
T[Total]	2046.91	1154.27	622.53	353.30
T[Step 3]	585.61	399.41	235.49	154.57
T[Step 5]	1442.22	742.90	377.89	188.59
[€] Total	-	0.887	0.822	0.728
€Step 3	- 1	0.733	0.621	0.473
Step 5		0.971	0.954	0.956
N[Step 3]	13250	8583	5785	4365

Table 3: Concurrent performance For 300×300 random points

ing: at intermediate times during the zero search, the nodes exchange a 'Zero Found Yet?' status flag, with all nodes breaking out of the zero search loop if any node returns a positive result.

For message passing machines such as the MarkIII, the checkpointing scheme is of little value, as the time spent in individual entries to Step 3 are not enormous compared to the node-to-node communication time. For example, for the 2-node solution of the 300×300 problem, the mean time for a single entry to Step 3 is only about 46 msec, compared to a typical node-to-node communications time which can be a significant fraction of a millisecond. The time required to perform a single Step 3 calculation is not large compared to node-to-node communications. As a (not unexpected) consequence, all attempts to improve the Step 3 efficiencies through various 'Found Anything?' schemes were completely unsuccessful.

The checkpointing difficulties for a message-passing machine could disappear, of course, on a shared memory machine. If the zero-search status flags for the various nodes could be kept in memory locations readily (i.e., rapidly) accessible to all nodes, the problems of the preceding paragraph might be eliminated. It would be interesting to determine whether significant improvements on the (already good) efficiencies of the concurrent Munkres algorithm could be achieved on a shared memory machine.

References

- F. Burgeios and J. C. Lassalle, 'An Extension of Munkres Algorithm for the Assignment Problem to Rectangular Matrices', Comm. of the ACM, 14(1971)802.
- 2. H. W. Kuhn, 'The Hungarian Method for the Assignment Problem', Naval Research Logistics Quarterly, 2(1955)83.

3. S. S. Blackman, Multiple-Target Tracking with Radar Applications, Dedham, MA: Artech House(1986).

Multi-Tiered Algorithms for 2-Dimensional Bin Packing*

Richard Fenrich State University of New York Buffalo, New York 14260 fenrich@cs.buffalo.edu Russ Miller
State University of New York
Buffalo, New York 14260
miller@cs.buffalo.edu

Quentin F. Stout University of Michigan Ann Arbor, Michigan 48109 qstout@zip.eecs.umich.edu

Abstract

This research is concerned with approximation algorithms for NP-hard optimization problems on hypercube multiprocessors. We investigate methods of solving such problems, focusing on the tradeoffs in running time, number of active nodes, input size, and accuracy of solution. In this paper, we consider a tiered algorithm framework that describes our level algorithms and we expand upon the 2-dimensional bin packing results given in [4]. The major contributions of this paper are data structures which dramatically improve the run time of the first fit and best fit algorithms presented in [4]. The results in this paper were obtained on a 32 node Intel iPSC/2.

Introduction

A variety of important industrial optimization problems are known to be NP-hard, which implies that we should not expect to find efficient (i.e., polynomial time) algorithms yielding optimal solutions to these problems for all input sets. (These problems include packing items on trucks, scheduling jobs on a computer system, and a variety of stock-cutting problems, to name a few.) In fact, if $P \neq NP$ then even a polynomial number of processors (i.e., polynomial in the size of the input) cannot be used to produce efficient solutions to NP-hard problems.

For NP-hard problems, researchers typically study approximation algorithms which attempt to find a nearly optimal solution in an acceptable amount of time. Recently, this study has included algorithms for multiple processor machines. Parallel approximation algorithms for the traveling salesperson problem are given in [12, 5], for the 0/1 knapsack problem in [10], for the 2-dimensional bin-packing problem in [4], and for the multiprocessor scheduling problem in [3].

This paper focuses on the 2-dimensional bin packing problem, which is often referred to as the rectangle packing problem. The 2-dimensional bin packing problem consists of a set of orthogonal rectangles, with each

rectangle p_i having height h_i and width w_i , and a vertical strip V of width C. The objective is to pack the rectangles into V so as to minimize the height of V.

Two interpretations of 2-dimensional bin packing help illustrate the applicability of our work. First, if we do not allow rotations of the rectangles, then we can interpret the problem as minimizing the completion time of a computational system where rectangles correspond to program tasks. The height of a rectangle corresponds to the amount of processing time required and the width corresponds to the amount of memory required. Notice that in this situation C represents the total memory of the system that is available. Obviously, in this situation rotations do not make sense since memory cannot typically be traded for processing time. The second application is to stock-cutting where "raw" material comes in rolls from which we wish to cut out rectangular patterns. The waste of the raw material is minimized if we minimize the length of the strip used. In this situation it may be reasonable to allow the rectangles to be rotated by ninety degrees. One common characteristic of these applications is their ability to be considered in a dynamic sense in which rectangles are input in a stream or in a static sense in which we know the entire rectangle set prior to packing. The former case is known as 'on-line' packing while the latter is known as 'off-line' packing.

Due to the economic importance of efficient stockcutting, a wide variety of heuristic methods have been developed for these problems over the last 20 years. (The reader is referred to [1] for an excellent overview of bin-packing problems.) These algorithms include "level" (or "shelf" or "strip") algorithms, which allow one to apply knowledge gained from the 1-dimensional bin-packing problem to the 2-dimensional case. In this paper, we propose a multi-tiered algorithm framework that promotes modularization for a variety of these level algorithms. The first tier of the framework is responsible for any preprocessing of the rectangles while the second tier packs the rectangles in a sequential manner. The third and final tier post-processes the packing to improve the packing from the second tier. Straight forward divide-and-conquer techniques are used to implement this framework in a hypercube environment.

^{*}This work was partially supported by NSF grants IRI-8800514 and ASC-8705104.

The next section of the paper proposes the tiered computational framework for our level algorithms. The two ensuing sections reiterate some of the conclusions made in [4] and discuss implementation details on the iPSC/2. The fifth section investigates enhancements to the algorithms used in [4]. These enhancements include data structure improvements and increased attention to the preprocessing step. Finally, we present our conclusions and some final comments.

Algorithm Framework

All of our bin-packing approximation algorithms are based on the concept of level algorithms [2]. Such level algorithms, including many of those presented in [1], can be considered in the following three-tiered framework:

- L1: Preprocess the rectangles.
- L2: Pack the rectangles by levels with each rectangle being placed so that its bottom rests on one of the levels. The levels are determined by the following constraints:
 - The bottom of the first level is the bottom of the vertical strip V.
 - Subsequent levels are determined by a horizontal cut through the top of the tallest rectangle in the previous level.

L3: Post-process the resultant packing.

The objective in each of the three tiers of this framework is to maximize the benefits from the time/quality tradeoff perspective. Results in [4] consider the benefits in performing certain preprocessing steps. For instance, heuristics that rotate the rectangles prior to packing improved packing efficiency dramatically in certain cases while impacting insignificantly on the running time. Likewise, heuristics used in tiers L2 and L3 impact upon the quality of the final packing as well as the running time of the algorithm. Since this framework has been developed so that we may consider off-the-shelf 2-dimensional bin packing algorithms, it lends itself to modular descriptions of the component algorithms used in each tier.

We now consider each of the three tiers in turn by describing the possible computations in each tier. Consider a total of N rectangles as input to any tiered 2-dimensional bin packing algorithm. Let \mathcal{P} be a partition of the N rectangles into P subsets of N/P rectangles each. For each preprocessing algorithm the asymptotic running time within each partition of \mathcal{P} will be given as a function of N/P, the number of rectangles in each class of the partition.

Tier L1: Preprocessing

- P1: No preprocessing. These algorithms can be considered as 'on-line' algorithms. Time: Θ(1).
- P2: A sort, keyed by height, in each partition of \mathcal{P} . Time: $\Theta(N/P)$ assuming that the height of the rectangles is bounded by a constant.
- P3: A rotation of the rectangles so that their height is greater than or equal to their width. Time: $\Theta(N/P)$.
- P4: A rotation of the rectangles so that their width is greater than or equal to their height. Time: $\Theta(N/P)$.

In the case that the preprocessing occurs 'osl-line', the number of alternative preprocessing algorithms is bounded only by the number of one-to-one and onto mappings from the input rectangle set onto itself.

Tier L2: Packing

In this packing tier we have considered three fundamental algorithms in conjunction with two heuristics. The three level packing algorithms follow. The asymptotic running times listed reflect the analysis considered in [4] where P corresponds to the number of active nodes and N equals the total number of rectangles to be packed.

- A1: Next fit packs rectangles left justified in the remaining unused width of the current level. If a rectangles will not fit in the current level then a new level is initialized with this rectangle and the packing continues with the new level assuming the role of the current level. Time: $\Theta(N/P)$.
- A2: First fit packs rectangles left justified into the remaining unused width of the lowest level that they will fit in. If a rectangle wil not fit in any of the existing levels then a new level is initialized with this rectangle. Time: $\Theta(N^2/P^2)$.
- A3: Best fit packs rectangles left justified into the remaining unused width of a level they fit in that minimizes the unused width of all such levels. If a rectangle will not fit in any of the existing levels then a new level is initialized with this rectangle. Time: $\Theta(N^2/P^2)$.

The two heuristics involved in the packing process were considered in [4]. Since these methods depend on both the distribution of the data and the complex relationships between levels, an asymptotic time analysis seems inappropriate.

- H1: Level Combining. When two levels, call them L_1 and L_2 , are complete we consider combining them in a simple fashion so as to reduce the total height taken by the two. This combination procedure consists of taking one of the levels, say L_1 , reversing the rectangles (i.e., the leftmost rectangle becomes the rightmost, etc.), moving the rectangles to the top of the level, and then lowering the newly rearranged L_1 on top of L_2 as far as possible.
- II2: Width Covering/Level Unpacking. When a level L is complete, we will unpack the rectangles in L if the unused width of L is greater than some heuristic factor. All unpacked rectangles will be repacked at the end of the algorithm by using a post-processing algorithm.

Tier L3: Post-processing

The post-processing tier has been considered in two possible ways.

- O1: If there are any rejected rectangles from step L2 these rectangles are accumulated and packed using some combination of the packing steps in L1 and L2.
- O2: No post-processing.
- The possible operations that can be used in this tier are considerably more complex than those in tier L1. In fact, the set of possible operations could consider interactions between levels as well as interactions between individual rectangles.

Parallel Realization

The framework presented above can be applied when using hypercube computers. In [3], a variety of hypercube solutions were given for solving the multiprocessor scheduling problem. These algorithms can be viewed as instances of a bottom-up parallel divide-and-conquer solution strategy, where initially each node solves the problem on its own set of data, followed by a sequence of steps where these partial solutions are combined to give the complete solution. Our level algorithms follow this basic approach. A level algorithm template that incorporates the multi-tiered framework is given below. We assume that given an input set of N rectangles, each of the P nodes of the hypercube initially assumes responsibility for N/P rectangles.

Rectangle Packing

1. Preprocessing can occur at a local level when the partition, \mathcal{P} , of the rectangles is the partition induced by the nodes. Preprocessing can occur at

- a global level when we view the partition as composed of the single set of all N rectangles.
- 2. Every node uses a level packing algorithm possibly augmented by a heuristic to independently pack its rectangles into a vertical strip of width C.
- 3. Recursive doubling is used to combine the independent solutions into a global solution. Post-processing heuristics are applied in this stage.

Previous Results

Several combinations of the tiers L1, L2, and L3 were considered in [4]. In the case that rectangle rotations were allowed (P3 and P4) we concluded that if time is critical, then

- the most efficient packing of the rectangles is by the next fit algorithm, as follows.
 - If the number of rectangles per node is relatively small (e.g., 8 or fewer), then include the preprocessing heuristic P4.
 - Otherwise, include the width covering and level combining heuristics, H1 and H2.

Conversely, if time is not as critical, then

- the first fit algorithm should be used as follows.
 - If there is a relatively small number of rectangles per node then widthwise rotating and the level unpacking heuristic should be used.
 - Otherwise, the heightwise rotation of rectangles should be used.

Next, we considered the packings for which rectangle rotations are prohibited. If time is critical, then

 the most efficient packing of the rectangles is using the next fit algorithm with the level unpacking and level combining heuristics.

Conversely, if time is not as critical, then

- the first fit algorithm should be used as follows.
 - Given no more than 32 rectangles per node, include the level unpacking heuristic.
 - For more than 32 rectangles per node, use the straight first fit algorithm.

Implementation Details

Initially, all nodes know the width, C, of the vertical strip and the initial seed for a random number generator. When the program begins, the host broadcasts the total number of rectangles to be packed to every node. It should be noted that all nodes know the same initial seed to the random number generator so every node can generate a distinct set of random rectangles. We use the minimal standard generator, as described in [11], where if the i^{th} node is to generate k rectangles, then it uses 2k random numbers beginning at random number 2k(i-1)+1. We store the rectangles in a static array that holds the maximum number of rectangles that will ever be used in the node.

After every node has generated its rectangle set using the random number generator, the set of active nodes synchronize. Each node continues by sampling the clock and by using a three-tiered algorithm to pack the rectangles. Upon completing a level during packing, the node accumulates the packing statistics for that level. When the recursive doubling step is performed the packing heights from each node are collected. In addition, if this step is a nodes last operation in the recursive doubling procedure the clock is sampled again, the running time of this node is determined and this time is sent to a neighboring node in the recursive doubling procedure. Of course, when heuristic H2 is used, the unpacked rectangles are retained and passed in the recursive doubling step along with the other relevant packing statistics. The running time of an algorithm is simply the maximum running time of all nodes.

Performance Analysis

In this section we discuss the performance of algorithms that extend previous results found in [4]. Since we use randomly generated rectangles as input, it is not possible (in the sense that the problem is NP-hard) to determine the optimal packing. Therefore, if an algorithm A packs the rectangles into vertical strip V (which has width C) using height D, then we use the percentage of the area CD that the rectangles cover as a measure of the quality of the solution produced by A. We ran our algorithms on inputs of size $32,64,128,256,\ldots,1048576$, using 1,2,4,8,16, and 32 nodes. Our results consider rectangles with height and width independent and uniform on (0...C].

Next Fit

The next fit decreasing height algorithm, one variation of next fit with the P2 heuristic, was previously explored in [4]. This research considers the next fit algorithm with the heuristics P2 and O2 where P2 is implemented as a global sorting routine. This global

sort was realized as a local sort followed by a merging operation. In the merging operation each node would route sets of rectangles that are grouped by height to the nodes responsible for rectangles of those particular heights. Upon receiving a set of rectangles, the set is merged in sorted order into the current set of rectangles on the node. In general, as the same number of rectangles are spread across more processors this method performs better than the next fit decreasing height of [4]. The packing improvement is the highest when using 32 nodes and is generally more than 1%. Alternatively, this method always performed 5% worse than the next fit decreasing height algorithm with level unpacking ([4]) and many times it packed with 10% less efficiency.

Interestingly, given a fixed number of rectangles greater than 1024, the packing efficiency was nearly identical for every number of active nodes. In fact, when using P and Q nodes, where $P \neq Q$, the packing efficiency differs by at most 0.3% in every case where the total number of rectangles is larger than 1024. This phenomenon is presented in Figure 1 and is explained as follows. The global sort distributes the rectangles in ordered intervals across all nodes. Given P nodes and N rectangles, the packing in the P nodes will differ only slightly from the one in 2P nodes. In both cases the first N/2 rectangles will be packed identically. The packing for the second half of the rectangle set in 2P nodes differs from the packing in P nodes by at most the height of the level that contains the $(N/2+1)^{st}$ rectangle. For a large number of sorted rectangles the height of any two neighbors is nearly identical and hence the level heights remain nearly the same. The dominance of the time in the higher number of nodes is attributed to the overhead associated with the global sort.

First Fit

A major improvement in the computation time of first fit with local pre-sorting was achieved. We implemented a static tree structure in which the leaf nodes were heaps. Each heap represents all of the levels with a particular amount of unused width. The root of each heap contains the minimum layer number with that particular amount of space left. The search for the lowest indexed level that will fit an input rectangle starts at the lowest indexed leaf node that can possibly fit the rectangle. By traversing the tree from leaf level to root level and using information stored in the trees nodes the correct level can be identified. For 32,768 rectangles per node our implementation of first fit decreasing height with a static tree was about 18.8 times faster than the first fit decreasing algorithm used in [4]. For the smaller number of rectangles the overhead associated with the static tree dominated the run time. In the worst case, the static tree implementation was 6.5 times slower than that of [4] with the static tree using 58 milliseconds and the older implementation taking 9 milliseconds. This case appeared for 32 total rectangles on one node.

First fit augmented with global version of heuristic P2 was implemented next. Similar effects to those of next fit with the global presorting were noticed. But, the effect was not nearly as dramatic since neighboring rectangles in the sorted order do not necessarily belong to the same or neighboring levels in the packing. In addition to this effect the preprocessing also increased the packing efficiency over that of first fit decreasing height for any given number of rectangles and more than one node. The increase in packing efficiency was over 1% many times but never more than about 2.2%. This effect is best explained by the fact that any given node will have a contiguous set of sorted rectangles. Thus, in a first fit packing less space will be wasted by placing relatively short rectangles into relatively tall levels since the heights are closer in proximity to each other than in the first fit decreasing case.

Best Fit

Best fit decreasing height had similarly striking improvements in its data structure. Two possible improvements were investigated. The first improvement was the use of the static tree structure that we used for first fit. The only difference in the first fit and best fit implementations is in tree search methods. In the best fit decreasing height implementation the lowest indexed leaf node that contains a level with enough space to fit the next rectangle is chosen. Therefore, the implementation turned out slightly faster than that of first fit in most instances since the amount of search logic has been reduced. In the best case, this implementation was about 167 times faster than the best fit decreasing height algorithm presented in [4].

The second time saving measure was to implement the best fit decreasing height by using balanced search trees [13] instead of static trees. The implementation of this structure improved the solution in two respects. First of all, we were no longer bound to the implementation dependent static tree and secondly, the tree traversal overhead associated with the static tree was minimized. Thus, this implementation of best fit ran at least 9% faster than best fit decreasing height with static trees. The largest speedup was achieved for smaller number of rectangles because the overhead on the static tree implementation dominated the timing. Figure 2 presents the timings and performance results using a balanced search tree.

Comparison

With respect to the algorithm comparison presented

in [4] several modifications can be made. We are now in the position to suggest the use of a best fit algorithm in place of the corresponding first fit algorithm for several reasons. As we have pointed out earlier, the balanced search tree implementation is preferred since it is faster, more elegant and more flexible than the static tree implementation. Additionally, as we mentioned in [4] best fit and first fit are nearly comparable in their packing efficiency. The time critical suggestions made in [4] remain valid with the realization that now there are fast implementations of best fit. Application writers may find that the advantages of a much better packing efficiency with a slower speed outweigh the advantages of the fast but rather inefficient packing with next fit. The last modification is the use of best fit in the cases in which P3 and P4 are prohibited. The best packings in these cases will be given by best fit augmented with the global presorting heuristic in the case that there are more than 32 rectangles per node. Otherwise, if there are less than 32 rectangles per node use best fit with heuristic II2.

Final Remarks

In this paper, we considered a multi-tiered framework for 2-dimensional bin packing algorithms. This three-tiered framework describes many level algorithms and is extensible enough to include many other algorithms not considered here. The emphasis in each of these three tiers is on the optimization of the packing quality/packing time tradeoff. The tiers describe modular algorithms.

We included new results to complement those presented in [4]. All algorithms were implemented for input sets with between 32 and 1048576 rectangles on an Intel iPSC/2 with between 1 and 32 active nodes. In particular, we considered the effects of global presorting as well as improved data structures. Conclusions were drawn incorporating the new results.

Our future plans include the consideration of relationships between the width of the vertical packing strip and the sizes of the rectangles. For example, it may be interesting to consider the heights and widths of the rectangles chosen independently and uniformly on (0...bC], for b < 1. In addition, we also plan to consider 'on-line' algorithms in depth. These algorithms typically use the preprocessing heuristic P1. We would also like to study various post-processing operations and the theoretical bounds placed on level packing algorithms by the various tier components.

References

[1] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson, Approximation algorithms for bin-packing - An updated

- survey, Bell Laboratories, Murray Hill, New Jersey, Technical Report, 1988. (This is an update of [7].)
- [2] E.G. Coffman, Jr., M.R. Garey, D.S. Johnson, and R.E. Tarjan, Performance bounds for level-oriented two-dimensional packing algorithms, SIAM J. Comput., 1980, pp. 808-826.
- [3] E. Cohen and R. Miller, Hypercube algorithms for the multiprocessor scheduling problem, Supercomputer (27), vol. V, no. 5, September, 1988, pp. 17-32.
- [4] R. Fenrich, R. Miller, and Q. Stout, Hypercube Algorithms for some NP-Hard Packing Problems, Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications, 1989, to appear.
- [5] E.W. Felten, Best-first branch-and-bound on a hypercube, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, pp. 1500-1504.
- [6] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Co., San Francisco, 1979.
- [7] M.R. Garey and D.S. Johnson, Approximation algorithms for bin-packing A survey, in Analysis and Design of Algorithms in Combinatorial Optimization, G. Ausiello and M. Lucertini (eds.), Springer-Verlag, New York, 1981, pp. 147-172.

- [8] R.L. Graham, The combinatorial mathematics of scheduling, Scientific American, Mar. 1978, pp. 124-132.
- [9] D.E. Knuth, The Art of Computer Programming, Volume 3 / Sorting and Searching, Addison-Wesley Publishing Company, Reading, Mass., 1973.
- [10] J. Lee, E. Shragowitz, and S. Sahni, A hypercube algorithm for the 0/1 knapsack problem, Proceedings of the 1987 International Conference on Parallel Processing, pp. 699-706.
- [11] S.K. Park and K.W. Miller, Random number generators: good ones are hard to find, Communications of the ACM, Oct. 1988, pp. 1192-1201.
- [12] N. Toomarian, A concurrent neural network algorithms for the traveling salesman problem, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, 1988, pp. 1483-1490.
- [13] N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

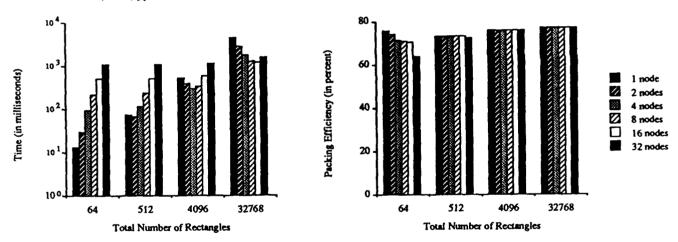


Figure 1: Next fit with a global pre-sort.

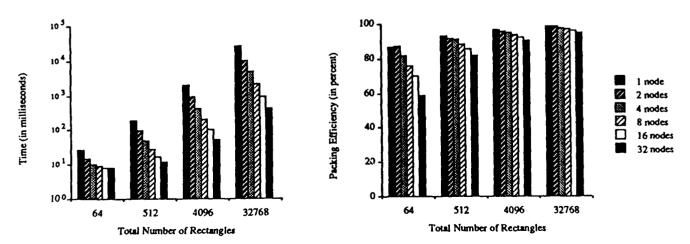


Figure 2: Best fit with a balanced search tree.

Efficient Serial and Parallel Subcube Recognition in Hypercubes

S. Al-Bassam*, H. El-Rewini**, B. Bose*, and T. Lewis*

* Oregon Advanced Computing Institute (OACIS) and
Department of Computer Science
Oregon State University
Corvallis, OR 97331

** Department of Math and Computer Science
University of Nebraska at Omaha
Omaha, NE 68182

Abstract

We develop an efficient subcube recognition algorithm that recognizes all the possible subcubes. The algorithm is based on exploiting more subcubes at different levels of the buddy tree. In exploiting the different levels, the algorithm checks any subcube at most once. Moreover, many unavailable subcubes are not considered as candidates and hence not checked for availability. This makes the algorithm fast in recognizing the subcubes. The number of recognized subcubes, for different subcube sizes, can be easily adjusted by restricting the search level down the buddy tree. The previous known algorithms become a special case of this general approach. When one level is searched, this algorithm performs as the original buddy system. When two levels are searched, it will recognized the same subcubes as the ones in [4] with a faster speed. When all the levels are searched, a complete subcube recognition is obtained. In a multi-processing system, each processor can execute this algorithm on a different tree. Using a given number of processors in a multi-processing system, we give a method of constructing the trees that maximizes the overall number of recognized subcubes. Finally, we introduce an allocation method "best fit" that reduces hypercube fragmentation. Simulation results and performance comparisons between this method and the traditional "first fit" are presented.

1. Introduction

Hypercube multiprocessors have been drawing considerable attention due to their structural regularity for easy construction and high potential for parallel execution [1-6]. Efficient allocation and/or deallocation of node processors in the hypercube is a key to its performance and utilization. The main objective is to maximize the utilization of the available resources as well as minimize the inherent system fragmentation. In this paper we introduce an efficient algorithm to achieve this objective. This problem has been studied in [1-4] using strategies

based on buddy and gray code systems. We propose two ways to extend the buddy system methodology to maximize the number of recognized subcubes with low fragmentation: 1) an extended buddy tree and 2) multiple extended buddy trees which is suited for a multi-processing system. Also we introduce a new heuristic for increasing the availability of the processors.

The buddy system algorithm searches for a subcube of dimension k only at level n-k of the buddy tree. The extended buddy tree algorithm generalizes the search for available subcubes to cover more than one level down the buddy tree. When the search covers all the levels beyond n-k (i.e. from n-k to n), complete subcube recognition can be achieved. At every search level, the proposed algorithm recognizes the candidate subcubes at that level only so that subcubes recognized at previous levels are not checked again. Also, it avoids checking many unavailable subcubes. Therefore, this algorithm performs significantly faster than exhaustively searching all subcubes at that level. The method becomes faster as the number of searched levels increases.

The multiple extended buddy trees method takes advantage of parallel computers. A set of disjoint trees are created and distributed among concurrently running processors. Each processor executes the extended buddy tree algorithm on its own buddy tree. It is desired to have different recognized subcubes at each processor to maximize the total number of recognized subcubes. Chen et. al.[1] has studied the distribution of multiple trees with depth 0 among many processors to maximize the recognized subcubes. In section 3, we consider the general problem of distributing the buddy trees with arbitrary depths to the different processors.

A hypercube is said to be fragmented when there are enough processors to accommodate a subcube request but they don't form a subcube. We study an allocation strategy that increases large subcube availability. When many candidate subcubes are recognized, the one with the minimum effect on the larger subcubes is allocated. In

section 4, this (best fit) method is discussed and compared to the traditional first fit method. Finally, in section 5 we give some concluding remarks.

We first introduce some notations that are used in this paper. Let Q_n denote a hypercube of dimension n and Σ be the ternary symbol set $\{0,1,*\}$, where * is a don't care symbol. Every Q_k subcube in Q_n can be represented by a string of symbols in Σ having k *'s. For example, the address of the subcube Q_2 formed by nodes 0010, 0011, 0110, 0111 in Q_4 is 0*1*. Let $\alpha*\alpha*$ represent the subcubes 0*0*, 0*1*, 1*0*, and 1*1*. In general, a string of length n of symbols in $\{\alpha,*\}$, where α is 0 or 1, with k *'s and n-k α 's represents 2^{n-k} subcubes of size k since α can be 0 or 1.

2. Extended Buddy Tree

In the buddy system [1], a binary tree of n levels, as shown in Figure 1 for n=4, is used to represent the availability of some subcubes of Q_n . In the buddy system, Q_k subcubes are recognized only at level n-k. There will be 2^{n-k} recognizable subcubes at level n-k, namely $\alpha\alpha...\alpha^{**}...^*$ with n-k α 's and k *'s, i.e. the subcubes α^{n-k} . For example, the Q_3 's at level 1 are 0^{***} and 1^{***} .

By extending the search to more levels (levels n-k, n-k+1, ... etc.), more subcubes can be recognized. When the next level is searched, i.e. level n-k+1, then $\binom{n-k+1}{n-k} 2^{n-k}$ subcubes can be recognized. This is true since any n-k bits can be chosen among the n-k+1 bits in the tree, recognizing the following:

$$\leftarrow \begin{array}{c} \leftarrow n - k + 1 \rightarrow \\ \alpha \alpha \dots \alpha^{**} \dots^{*} \\ \alpha \alpha \dots \alpha^{*} \alpha^{*} \dots^{*} \\ \alpha \dots \alpha^{*} \alpha \alpha^{*} \dots^{*} \\ & \vdots \\ & \vdots$$

i.e. the subcubes { $\alpha^{n-k-i} * \alpha^i *^{k-1} \mid 0 \le i < n-k$ }.

Using a similar argument, searching to depth d yields $\binom{n-k+d}{n-k} 2^{n-k}$ recognized Q_k subcubes, as stated in following Lemma.

Lemma 2.1: Let N(k,d,p) denote the number of recognizable Q_k subcubes between levels n-k and n-k+d, i.e. at depths 0 to d, where $0 \le d \le k$ using p trees (in this

section p = 1). Then in a hypercube of dimension n, $N(k,d,1) = \binom{n-k+d}{n-k} 2^{n-k} \quad \text{for } 0 \le d \le k. \quad \Box$

It is clear from lemma 2.1 that increasing the search depth (d) will increase the number of recognizable subcubes. Let D_k represent the maximum depth used in searching for Q_k subcubes, where $0 \le D_k \le k$. Also let Time(k,d) denote the number of comparisons needed to search for a Q_k subcube using one tree between levels n-k and n-k+d where $0 \le d \le k$. Notice that $N(k,k,1) = \binom{n}{n-k} 2^{n-k}$, which is all the possible subcubes of size k. The number of Q_k subcubes recognized at depth d (but not at the previous depths) is $N(k,d,1) - N(k,d-1,1) = \binom{n-k+d-1}{n-k-1} 2^{n-k}$. The maximum search time occurs when all the recognizable subcubes are not available and the search is forced to depth D_k . Therefore, $Time(k,D_k)$ can be bounded as shown in the following Lemma.

Lemma 2.2: Given a hypercube of dimension n, the search time needed to recognize a Q_k subcube using depth parameter D_k , is

$$\begin{split} \text{Time}(\textbf{k},\,\textbf{D}_{\pmb{k}}) \leq & \sum_{\textbf{d}=\textbf{0}}^{\textbf{D}_{\pmb{k}}} \, \left(\begin{smallmatrix} \textbf{n}\textbf{-}\textbf{k}+\textbf{d}-1 \\ \textbf{n}\textbf{-}\textbf{k}-1 \end{smallmatrix} \right) \, \, 2^{\textbf{n}\textbf{-}\textbf{k}+\textbf{d}} \, \leq \\ & \left(\begin{smallmatrix} \textbf{n}\textbf{-}\textbf{k}+\textbf{D}_{\pmb{k}} \\ \textbf{n}\textbf{-}\textbf{k} \end{smallmatrix} \right) \, 2^{\textbf{n}\textbf{-}\textbf{k}+\textbf{D}_{\pmb{k}}} \end{split}$$

Proof: When the search goes through all depths, i.e. $0,1, ..., D_k$, the time will be

$$\begin{aligned} & \text{Time}(k, \, D_k) \, \leq \, \sum_{d=0}^{D_k} (N(k, d, 1) - N(k, d-1, 1)) \, 2^d \\ & = \, \sum_{d=0}^{D_k} \left(\begin{array}{c} n \text{-} k + d \text{-} 1 \\ n \text{-} k \text{-} 1 \end{array} \right) \, 2^{n-k} \, 2^d, \, \text{and} \, \, \sum_{d=0}^{D_k} \left(\begin{array}{c} n \text{-} k + d \text{-} 1 \\ n \text{-} k \text{-} 1 \end{array} \right) \, 2^{n-k} \, 2^d \\ & \text{is bounded by} \, \left(\begin{array}{c} n \text{-} k + D_k \\ n \text{-} k \end{array} \right) 2^{n-k+D} k. \quad \Box \end{aligned}$$

As seen from Lemmas 2.1 and 2.2, both the number of recognized Q_k subcubes and the maximum search time increase as the depth parameter D_k increases. Figure 1 shows the increase in both log maximum time and log the number of recognized Q_6 subcubes in Q_8 . In general, table 1 illustrates the spectrum of the recognition and its maximum time.

Depth	Number of recognized Q _k 's	Maximum search time	Comment
0	2 ^{n-k} 2 ^{n-k+1}	2 ^{n-k} 2 2 ^{n-k+1}	Buddy system Gray code
1	$\binom{n-k+1}{n-k} 2^{n-k}$	$2 \binom{n-k}{n-k-1} 2^{n-k}$	·
D _k	$\binom{n-k+D_k}{n-k} 2^{n-k}$	$\sum_{d=0}^{D_k} \binom{n-k+d-1}{n-k-1}$	2 ^{n-k+d}
k	$\binom{n}{n-k} 2^{n-k}$	$\sum_{d=0}^{k} {n \cdot k + d \cdot 1 \choose n \cdot k \cdot 1}$ (full	2 ^{n-k+d} recognition)

Table 1.

Example 2.1: In a hypercube of dimension 4, let $D_0=0$, $D_1=0$, $D_2=2$, $D_3=1$, and $D_4=0$. Using the tree shown in Figure 1, we can recognize $N(k,D_k,1)=\binom{n-k+D_k}{n-k}2^{n-k}$ Q_k 's, for $0 \le k \le n$. That is, we can recognize $16 \ Q_0$'s, $8 \ Q_1$'s, $24 \ Q_2$'s, $4 \ Q_3$'s, and $1 \ Q_4$. The maximum search time for recognizing a Q_2 , for instance, is $\sum_{d=0}^2 \binom{1+d}{1}$ $2^{2+d}=4\times 1+8\times 2+12\times 4=68$ node comparisons. \square

To recognize the Q_k subcubes efficiently at depth d, i.e. at level n-k+d, we take into account two factors

- 1) It is not necessary to check the subcubes that have been checked at the previous levels.
- 2) Narrow down the recognition to 'candidate' subcubes instead of exhaustively checking all possible $\binom{n-k+d-1}{n-k-1}$ 2^{n-k} subcubes at depth d.

A Q_k can be formed by 2^d Q_{k-d} subcubes at level n-k+d. Any Q_k having an '*' in bit number n-k+d is generated at some level before n-k+d. At level n-k+d, we only generate subcubes that have no '*' in bit n-k+d. Only n-k-1 α 's can be chosen among the n-k+d-1 positions to recognize $\binom{n-k+d-1}{n-k-1} 2^{n-k}$ subcubes at depth d.

The candidate Q_k subcubes at level n-k+d are generated as follows. Let q be a Q_{k-d} subcube at level n-k+d. If q is available then all the Q_k subcubes containing q are

considered as candidates. When q is not available all the Q_k subcubes containing q are not available and hence not candidates. The other 2^d - 1 Q_{k-d} 's, that form with q a candidate subcube, can then be tested for availability. This process speeds up the search considerably.

More formally, let $S_n=\{\ \star^n\ \}$, $S_{n-1}=\{\ 0\ \star^{n-1},\ 1\ \star^{n-1}\ \}$, $S_{n-2}=\{\ 0\ 0\ \star^{n-2},\ 0\ 1\ \star^{n-2},\ 1\ 0\ \star^{n-2},\ 1\ 1\ \star^{n-2}\ \}$,... etc. In general let $S_i=\alpha^{n-i}\ \star^i=\{\ 0^{n-i}\ \star^i,\ ...,\ 1^{n-i}\ \star^i\ \}$ for $0\le i\le n$, i.e. S_k is the recognized Q_k subcubes at level n-k. Let T be the buddy system tree of Q_n . The algorithm is then stated as follows.

Algorithm 2.1: Subcube Recognition.

Given n, k, and D_k , the algorithm recognizes $\binom{n-k+D_k}{n-k}$ 2^{n-k} subcubes of dimension k.

For d = 0 to D_k do

For each $q = v_1 v_2 v_3 ... v_n \in S_{k-d}$ such that T[q] = true (i.e. available) do

- 1.1 Let Q be the set of all Q_k subcubes that contain q. For each subcube $p \in Q$ do
- 1.2 If the other $2^d 1 Q_{k-d}$'s forming p are available then p is available, stop.

The set Q in step 1.1 can be formed by changing any d 0's to *'s in the first n-k+d-1 positions of q. Notice that if position n-k+d was included then all previously recognized subcubes will be formed, and hence this position is avoided. Since $q = v_1 \ v_2 \ v_3 \ ... \ v_n$ where $v_i = '*'$ for $n-k+d+1 \le i \le n$, so $Q = \{p = u_1 \ u_2 \ ... \ u_{n-k+d-1} \ v_{n-k+d-1} \ v_{n-k+d-1} \}$ where the number of *'s in p is k [i.e. there are d *'s in $(u_1 \ u_2 \ ... \ u_{n-k+d-1})$] and if $u_i = '*'$ then $v_i = 0$ }. In general, the d *'s can be chosen in any of the positions from 1 to n-k+d-1, however when only the positions containing 0's (or 1's) are considered, every Q_k at this level will be generated by exactly one Q_{k-d} .

In step 1.2, given any subcube $p \in Q$, where $p = u_1 u_2 ... u_{n-k+d-1} *^{k-d}$. The other $2^d - 1 Q_{k-d}$ subcubes that form p are obtained by enumerating the d *'s in the first n-k+d-1 positions of p. These Q_{k-d} subcubes (having the last k-d positions as *'s) can be directly checked at level n-k+d of the tree. \square

The number of recognized subcubes can be controlled by setting appropriate values of D_k for different subcube

sizes. For instance, when $D_k = 0$ for $0 \le k \le n$, this method performs as the original buddy system. When $D_k = 1$ for $0 \le k \le n$, this method recognizes the same subcubes as in [4] with greater speed. In this case, this algorithm performs faster than the one in [4], especially when the system is highly loaded, since many unavailable subcubes are not considered as candidates whereas in [4] all possible subcubes at that level are candidates. The gray code strategy is somewhere between depths 0 and 1 (more towards the 0). On the other extreme, when $D_k = k$, for $0 \le k \le n$, all the possible subcubes can be recognized.

In example 2.1 (using Figure 1), all the 24 Q_2 's are recognized since D_2 =2, i.e. searching depths 0, 1, and 2. The following tabulation shows the recognized (but not necessarily candidate) subcubes at each level of the tree.

d = depth level recognized # of subcubes recognized subcubes only at depth d

0 2
$$\alpha\alpha^{**}$$
 4
1 3 $\alpha^{*}\alpha^{*}$ and α^{*} 8 with bit 3 \neq *
2 4 $\alpha^{**}\alpha$, $\alpha^{*}\alpha^{*}$, $\alpha^{*}\alpha^{*}$ 12 with bit 4 \neq *

Example 2.2: Consider the hypercube of dimension 4 represented by the tree T in Figure 1 where the dark subcubes are occupied and the light ones are available. The following sequence illustrate how algorithm 2.1 proceeds to recognize Q_2 when $D_2 = 2$.

When d=0, the subcubes 00**, 01**, 10**, 11** are considered but they are all not available. When d=1, Since q = 000* is busy the subcubes 0*0* and *00* are skipped. Since q = 001* is available we consider the candidate set $Q = \{0*1*, *01*\}$. 0*1* = 001* and 011* but 011* is not available so 0*1* is not. *01* = 001* and 101* but both are available so *01* is available. \square

3. Multiple Extended Buddy Trees

In this section we study the performance of the extended buddy system when multiple trees are employed. In a multi-processing system these trees can be assigned to different processors to speed up the recognition process.

Let $[a_1, a_2, ..., a_n]$ denote a tree that splits at the first level according to bit number a_1 , then at the second level according to bit a_2 , ... and so on. Every bit position appears exactly once, i.e. $\{a_1, a_2, ..., a_n\} = \{1, 2, ..., n\}$. The tree shown in Figure 1, say T_1 , is a [1, 2, 3, 4]. Figure 2 shows another tree $T_2 = [4, 3, 2, 1]$.

Recall that, as in Lemma 2.1, given a tree $T = [a_1, a_2, ..., a_n]$, the number of recognized Q_k 's up to depth d is $\binom{n-k+d}{n-k}$ 2^{n-k} . These subcubes are of the form $V = v_1 \ v_2 \ \ v_n$ such that V has n-k α 's and k *'s and the α 's can only appear in the positions $a_1, a_2, ...,$ or a_{n-k+d} .

Let C_1 and C_2 be the sets of the recognized Q_k subcubes using $T_1 = [a_1, a_2, ..., a_n]$ and $T_2 = [b_1, b_2, ..., b_n]$ to depth d, respectively. C_1 and C_2 are distinct, i.e. $C_1 \cap C_2 = \emptyset$, iff $\{a_1, a_2, ..., a_{n-k+d}\} \cap \{b_1, b_2, ..., b_{n-k+d}\} \mid < n-k$. This is true since the n-k α 's in C_1 can never be in the same positions as the n-k α 's in C_2 .

For example, consider $T_1 = [1,2,3,4]$ in Figure 1. When $D_3=1$, the recognized Q_3 's at levels 1 and 2 are α^{***} and * α^{***} , and when T2 = [4,3,2,1] in Figure 2 is used, the recognized Q_2 's are ** α^* and *** α . The subcubes are disjoint since $I(1,2) \cap \{4,3\} \mid < n-k = 1$.

Distinct Trees Matrix (DTM):

A DTM for a hypercube of dimension n represented by m trees, is defined to be an m×n matrix, where each row represents a different tree.

Let DTM =
$$\begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ & & & & \dots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix}$$

where row i is a
$$[x_{i1}, x_{i2}, ..., x_{in}]$$
 tree, 'herefore $\{x_{i1}, x_{i2}, ..., x_{in}\} = Z_n = \{1, 2, ..., n\}$ for $1 \le i \le m$ (3.1)

A matrix M is considered a DTM if for any i distinct numbers from Z_n there is at least one row in M such that these i numbers appear in the first $i+Y_{n-i}$ positions in that row. When M has few rows, not all the permutations are achievable. In order to maximize the total number of recognized Q_k 's, the DTM must have the maximum possible permutations. More formally, let $S_{k,i}$ be the set of all possible subsets of $\{x_{i1}x_{i2}...x_{i(n-k)+Y_k)}\}$ of size n-k. Let $C_k = \{S_{k,j} | 1 \le j \le m\}$, i.e. C_k represents the set of the recognized Q_k subcubes. In the construction of the DTM, the C_k 's must be maximized, so a DTM must satisfy the following property.

$$|C_k| = \min \left(m \binom{n-k+Y_k}{n-k} \binom{n}{n-k} \right)$$
for $0 \le k \le n$ (3.2)

Example 3.1: Let n = 4, m = 6, and $Y_k=0$ for all k. The following 6×4 matrix is a DTM

1	2	3	4
1 2 3 4	2 3	3 4	1 2 3 4 3
3	4	1	2
4	1	1 2	3
1 2	1 3	2	4
2	4	1	3

Using the 6 trees (with depth 0) given in the above DTM allows us to recognize all possible subcubes in a hypercube of dimension 4. For example all the Q_2 subcubes are recognized because in columns 1 and 2 all the possible combinations exist. For instance, the $\alpha^{**\alpha}$ Q_2 subcubes are recognized from the [4,1,2,3] tree. \square

Example 3.2 Let n = 6, m = 3, $Y_k = 1$ for $0 \le k \le 6$. Then the following matrix is a DTM.

$$\mathbf{M} = \left(\begin{array}{cccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 1 & 2 \\ 5 & 6 & 1 & 2 & 3 & 4 \end{array}\right)$$

The trees in M give the maximum recognized subcubes. In this case, all the subcubes of size 0, 1, 5, and 6 are recognized; and the recognized subcubes of size 2, 3, and 4 are maximized. For instance, the ** α * α * Q_2 subcubes are recognized from the [3,4,5,6,1,2] tree since {3,5} appear in the first n-k+Y_k = 3 positions of this tree.

Let $Y = \max \{ Y_k \mid 0 \le k \le n \}$. A DTM of $\lfloor \frac{n}{Y+1} \rfloor$ or less rows can be formed by letting the first row be $\{1, 2, ..., n\}$ and row i be a Y+1 left circular shift of row i-1, for $1 \le i \le \lfloor \frac{n}{Y+1} \rfloor$. The matrix M in the previous example is generated using this method. When all the depths are 0 then a DTM with any number of rows can be constructed using matching theory [1]. In the presence of a multi-processing system with p processors, a DTM with p rows and n columns provides the best tree to processor assignment. That is, processor i performs the search using the tree in i^{th} row of the DTM, for $1 \le i \le p$.

Lemma 3.1: Given p processors, let M be a $p \times n$ DTM. The maximal subcube recognition is achieved by assigning the p trees of M to the p processors. And,

$$N(k,Y_k,p) = \min\left(\binom{n}{n-k} 2^{n-k}, p\binom{n-k+Y_k}{n-k} 2^{n-k}\right)$$

where $N(k,Y_k,p)$ is the total number of the recognized Q_k 's up to depth Y_k in the p trees of M. \square

As seen from Lemma 3.1, when each processor in the parallel system is assigned a different tree, the number of the recognized $Q_{\mathbf{k}}$'s is linearly dependent on the number of processors.

Lemma 3.2: Given an $p \times n$ DTM, M, and Y_k for $0 \le k \le n$. Let R_k denote the number of the recognized Q_k 's,

where
$$0 \le R_k \le \binom{n}{n-k} 2^{n-k}$$
, then
$$p \ge \frac{R_k}{\binom{n-k+Y_k}{n-k} 2^{n-k}} \quad \text{for } 0 \le k \le n.$$

Proof: From lemma 3.1, any tree can recognize $\binom{n-k+Y_k}{n-k}$ 2^{n-k} Q_k subcubes when searched to depth Y_k . Therefore, all the trees in the DTM can recognize at most $p\binom{n-k+Y_k}{n-k}$ 2^{n-k} Q_k subcubes when searched up to depth Y_k , i.e. $R_k \le p\binom{n-k+Y_k}{n-k} 2^{n-k}$. \square

Consider the following special cases of the Lemma 3.2.

1) Suppose that $R_k = \binom{n}{n-k} 2^{n-k}$ and $Y_k = 0$ for all k, i.e. complete recognition with zero depth. Then, $p \ge \frac{\binom{n}{n-k} 2^{n-k}}{\binom{n-k}{n-k} 2^{n-k}} = \binom{n}{n-k}$ for $0 \le k \le n$. The

maximum occurs when k = n/2, in which $p \ge \binom{n}{n/2}$.

2) Suppose that $R_k = \binom{n}{n-k} 2^{n-k}$ and $Y_k = k$ for all k. In this case $m \ge 1$ for $0 \le k \le n$. This confirms that a single tree can recognize all the subcubes when searched up to depth k.

Figure 3 illustrates, for n=8 and k=6, the relation between the search depth in each tree and the number of required trees needed to recognize all the $\binom{8}{8-6}$ 2^{8-6} Q_6 subcubes in Q_8 . The figure also shows the log of the maximum search time for the different depths.

As seen in section 2, a uni-processor can recognize

$$N(k, D_k, 1) = {n-k+D_k \choose n-k} 2^{n-k} Q_k \text{ is in at most}$$

$$\sum_{d=0}^{D_k} {n-k+d-1 \choose n-k-1} 2^{n-k+d} \text{ time units. Using p processors,}$$
the $N(k, D_k, 1) Q_k \text{ is can be recognized much faster since}$
the search depth (Y_k) in each processor is smaller than D_k . To recognize $N(k, D_k, 1) Q_k \text{ is using p trees, the trees}$
must be searched up to depth $Y_k \text{ where p * } N(k, Y_k, 1) \ge N(k, D_k, 1) \text{ i.e., p} {n-k+Y_k \choose n-k} 2^{n-k} \ge {n-k+D_k \choose n-k} 2^{n-k}.$
Using the approximation ${n-k+Y_k \choose n-k} \approx {n-k+D_k \choose n-k} Y_k - D_k$

$${n-k+D_k \choose n-k} \text{ we get}$$

$$Y_k \approx D_k - \frac{\log p}{\log n - \log k}$$
3.2.1

Let T_1 be the maximum time to recognize all possible Q_k subcubes in Q_n with one processor. Let T_p be the time to recognize all possible Q_k subcubes in Q_n using the best parallel algorithm on a parallel system of p processors, so $T_p = \frac{T_1}{p}$. Let T_p^* denote the maximum

time to recognize all possible Q_k subcubes using the multiple trees method with p processors. Let improve(p) = $\frac{Tp}{T}$ then T_p

improve(p)
$$\geq \frac{1}{p} \left(\frac{2n}{n-k}\right)^h$$

where $h = \frac{\log p}{\log n - \log k}$ 3.2.2

This can be proved as follows. The max time to recognize all Q_k 's using one tree is $T_1 = \binom{n}{n-k} 2^{n-k} 2^k$

and hence
$$T_p = \frac{\binom{n}{n-k} 2^n}{p}$$
. When p trees are used then

$$T_{p}^{*} = {n-k+y \choose n-k} 2^{n-k+y} \text{ where } y \approx k - \frac{\log p}{\log n - \log k} ,$$
 so
$$T_{p}^{*} \leq {n-k \choose k}^{k-y} {n \choose n-k} 2^{n-k+y}$$

then improve(p)
$$\geq \frac{1}{p} \left(\frac{2k}{n-k} \right)^h$$
 where $h = \frac{\log p}{\log n - \log k}$

The improvement is mainly due to the usage of the distinct multiple trees in T_p^* . So it is faster to distribute the trees among the processors rather than distributing the original function.

Also, notice that more than one tree can be used at each processor, and so all these distinct multiple trees can run on one processor. This will yield a faster subcube recognition; however, the major drawback of this method, other than increased memory, is the time taken to update the trees after each subcube allocation.

4. Subcubes Recognition with High Availability (Low Fragmentation).

In the sections 2 and 3 the emphasis was on the number of recognized subcubes. When low fragmentation is desired, an allocation strategy that chooses among the recognized subcubes must be employed. This problem is similar to the traditional memory system where the memory is allocated based on some strategy, say first fit, best fit, worst fit, etc. The fragmentation problem also extends to the deallocation, i.e. when a subcube becomes available. In this case one can also rearrange the processor-task mapping to minimize fragmentation. This is somewhat similar to compacting the memory.

The current allocation methods can be considered as first fit since the first available subcube is chosen for allocation. We propose a new method similar to the best fit in memory systems. This method chooses the subcube, among all available subcubes, that leaves the maximal unfragmented system. Simulation results and performance comparisons between this method and the "first fit" will be presented. We start with some definitions.

Let $A = \langle a_n, a_{n-1},, a_0 \rangle$ denote the subcube availability vector, where a_k is the number of available (and recognizable) Q_k subcubes. We call A the "state" of the system. The initial state is $\langle N(n,D_n,1), N(n-1,D_{n-1},1),, N(0,D_0,1) \rangle$. Let $A = \langle a_n, a_{n-1}, ..., a_0 \rangle$ and $B = \langle b_n, b_{n-1},, b_0 \rangle$ be two states and let j be the largest integer $(0 \le j \le n)$ where $a_j \ne b_j$. We say that A is less fragmented than B iff $a_j > b_j$, i.e. lexicographic order. This metric gives more weight to larger subcubes than smaller ones.

Let q be a recognizable and available subcube and let $L_q =$ <1_n, 1_{n-1}, ..., 1₀> be the loss vector resulting from

allocating q. The loss vector implies that $\mathbf{l_i}$ $\mathbf{Q_i}$, for $0 \le i \le n$, recognizable subcubes are made unavailable as a result of allocating q. The new state after allocating q will be:

new state = current state -
$$L_q$$

In order to achieve less fragmentation (high availability) when allocating a new subcube, we choose the one that causes the minimum loss to large subcubes in the allocation process. This is illustrated in the following algorithm.

Algorithm 4.1: Best Fit Allocation (using a single tree).

- o Let S be set of all the recognized and available Q_k 's.
- o Let $q \in S$ be the subcube such that $\ L_q \leq L_p$ for any $p \in S$.
- o Allocate q (if any).
- o The tree and the state are updated accordingly. \square

In the above algorithm, when all the depths are restricted to zero, L_q can be easily calculated by counting the number of Q_k 's that become unavailable at level n-k as a result of allocating q. When q is of dimension k, L_q will be of the form <0, ..., 0, 1, ..., 1, 2, 4, ..., 2^{k-1} , 2^k >. So L_q can be determined by the largest lost subcube corresponding to the first "1" in L_q . When the depths are arbitrary, the computation of L_q might take longer time.

A simulation was performed to analyze the new method. The following parameters were varied: hypercube sizes, load factors, and the sizes of the requested subcubes. For the later, the uniform and geometric distributions where used. Figure 4 is for Q_8 with system load between 80% to 100%. The geometric distribution was used to generate the size of the requested subcubes. At every time unit a random subcube is chosen from the allocated subcubes to be released. This gives a semi-exponential distribution for their execution time. The simulation was run for 1000 time units, after reaching a 90% load factor. Figure 4 suggests that for higher subcube sizes the best fit performs considerably better than the first fit method.

When multiple trees are used, let A_i be the state of the system using the tree at processor i, for $1 \le i \le p$. The system (global) state is then defined as the maximum (element wise) of the A_i 's. In this case, algorithm 4.1 is executed at each processor. Processor i then sends its new state A_i and its candidate subcube q_i , i.e. with the one with the lowest loss. The "host" collects this information and chooses the best among q_i 's. Algorithm 4.1 can be

modified for multiple processors (trees) as follows.

Algorithm 4.2: To recognize a Q_k subcube with low fragmentation using p trees.

- o Let q_i and A_i be the candidate subcube and the new state, respectively, of processor i.
- o Processor i sends its qi and Ai to the host.
- o The host computes m such that $A_m \ge A_i$ for $1 \le i \le p$.
- o Allocate q_m (if any).
- o Each processor updates its tree and state accordingly. \square

To completely remove system fragmentation, deallocation must be considered. When a task releases a given subcube, the cube might become fragmented. In this case one can rearrange the task-processors allocation to remove this fragmentation. This process, referred to as task migration [3], has a high overhead since it requires task deallocation and allocation. Task migration can be done at every deallocation, if fragmentation exist, to maintain an unfragmented system. However, when the system is highly available it may not be worthwhile. The other approach to this problem is to compact the whole system when the fragmentation exceed some threshold.

References

- [1] M. Chen and K. G, Shin, "Processor Allocation in an N-cube Multiprocessor Using Gray Codes", IEEE Transaction on Computers, Vol. C-36, No.12, 1987.
- [2] M. Chen and K. G. Shin, "Embedment of interesting task modules into a hypercube multiprocessor", Proc. Second Hypercube Conference, 1986.
- [3] M. Chen and K. G, Shin, "Task Migration in Hypercube Multiprocessor", to appear.
- [4] A. Dehlaan and B. Bose, "A New strategy for Processor Allocation in an N-cube Multiprocessor", Phoenix Conference on Computer and Comm., March 1989.
- [5] B. Becker and H. Simon, "How robust is the n-cube?", Proc, 27th Symp. on Foundations of Comp. Sci. 1986.
- [6] L. N. Bhuyan and D.P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network", IEEE Transaction on Computers, Vol. C-33, 1984.

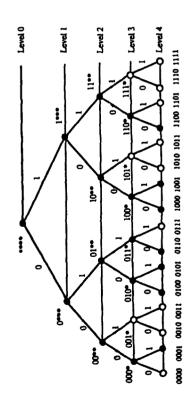


Figure 1. A Q_4 buddy tree (dark subcubes are occupied and light ones are available).

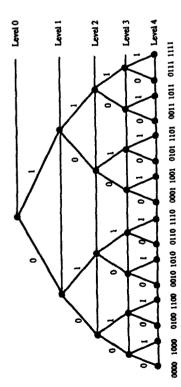


Figure 2: A [4,3,2,1] tree.

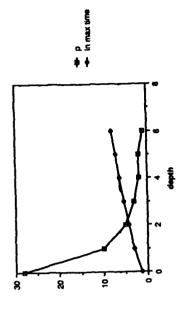


Figure 3. The log of the number of trees (and the log of the maximum time) needed to recognize all Q_6 subcubes in Q_8 when searched to depth d.

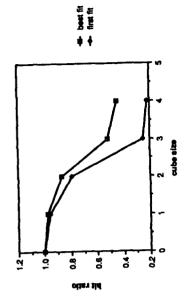


Figure 4. Hit ratio for different sizes of subcubes allocation using best fit and first fit allocation.

Parallel Thinning on a Distributed Memory Machine

Joong H. Back and Keith A. Teague

School of Electrical and Computer Engineering Oklahoma State University Stillwater, Oklahoma 74078

Abstract

A parallel thinning algorithm based on boundary following is presented in this paper. The boundary of each object region is extracted and linked in parallel. The resulting object boundary data is divided based on the object size and the number of nodes for load balancing, then the divided objects are redistributed to the nodes. Each boundary in a node is projected on a "working plane". Next, the boundary data is repeatedly shrunken until only the skeleton of the region remains. The conventional iterative parallel algorithm as well as our new algorithm are implemented on a hypercubetopology multiprocessor computer, the Intel iPSC/2. The two algorithms are compared and analyzed. Some resulting figures and execution times are presented.

1. Introduction

Thinning is one of the most important procedures in pattern recognition and image data reduction, but it is a very time consuming procedure. In most existing thinning algorithms, several templates (usually 3x3) are scanned on the image for deleting boundary points but not the skeleton of an object. This procedure is repeated until no points are deleted. Obviously the complexity of the whole procedure is $O(n^3)$ [1]. In order to reduce processing time many parallel algorithms [2-7] has been proposed which can be easily implemented on currently available mesh computers. Unfortunately, those parallel thinning algorithms are undesirable for implementation in distributed memory computers because the global shapes of the objects in an image might be affected when the image is divided and distributed to each node. To avoid this problem, data swapping between nodes, that is, communication, must be performed at every iteration [8].

In this paper, we propose a new parallel thinning algorithm based on boundary following and shrinking. Each object boundary in the image is extracted and linked in parallel. The number of objects is divided based on the number of nodes and object size. Then the objects are distributed to the nodes and thinned in parallel by following the boundaries and shrinking them in the direction perpendicular to the boundary and pointing toward the inside of the object. Ihis algorithm reduces the complexity from $O(n^2)$ to $O(n^2)$.

2. Parallel Boundary Detection and Object Extraction

To avoid the problem of breaking the objects between nodes, we extract the objects and thin them individually in each node in parallel. In this section we introduce a parallel boundary detection and object extraction method on a distributed memory computer.

2.1 Input Image Distribution Method

Poor performance can result if processor loading is uneven. In order to maximize the performance, the amount of data loaded to each node must be balanced. A distribution method in which the image plane is divided into rows as evenly as possible according to the number of nodes being used and each resulting sub-image is distributed to each node is commonly used in parallel image processing. It is obvious that the processing time can be reduced in theory to O(1/N) by using this distribution method, where N is the number of nodes being used. Here some rows on the borders of each node need to be shared with its adjacent nodes for detecting boundaries by using the template matching method described in the next section. The number of shared rows depends on the number of rows of the template: r/2 rows need to be shared, where "r" is the number of rows of the template used.

2.2 Boundary Detection

Let us assume that we have a digitized binary image. Then an object region in the image can be simply represented by the set of boundary points, or connected edge points, of the object. The connectedness of two boundary points in a binary image depends on the definition of the neighborhood: four-neighbor (N_{χ}) or eight-neighbor (N_{χ}) [9]. N_{χ} and N_{χ} neighborhoods of a point at (i,j) consist of the following points:

$$N_4 = ((i,j-1), (i,j+1), (i-1,j), (i+1,j))$$
 (1)
 $N_8 = (N_4, (i-1,j-1), (i-1,j+1), (i+1,j-1), (i+1,j+1))$ (2)

In this paper we have chosen eight-neighbor as the

definition of the neighborhood. Two points are eight-connected if they are eight-neighbors of each other. Figure 1 shows a 3x3 template according to the eight-neighbor definition.

Let us assume that the size of the input binary image is nxn, and no object points touch the periphery. Then the boundaries can be detected by the following procedure:

```
procedure Boundary_Detection;
begin
for j=0 to n-1 do begin
for i=0 to n-1 do begin
if I(i,j)=1 and I(p,q)≈1 for all
(p,q) ∈ Ng, then B(i,j)=0;
else B(i,j)=1;
end;
end;
end;
```

where I is an array representing the input image, and B is an array representing the output which contains only boundary points. This procedure is performed in each node for its sub-image in parallel.

Pe	Pi	P ₂
(i-1,j-1)	(i,j-1)	(i+1,j-1)
P ₇ (i-1,j)	Pa (i,j)	P3 (i+1,j)
P ₆ (i-1,j+1)	P5 (i,j+1)	P4 (i+1,j+1)

Figure 1. 3x3 template according to the eightneighbor definition

2.3 Object Extraction

As the result of the boundary detection, each node has boundary points only for its own sub-image. Now, we extract the objects locally by the boundary-following method which is described in [9] (see ch.4). The data structure for an object might be the following:

```
typedef struct {
   int numofbound, /* number of boundary
                   /* points
       numoftbe,
                    /* number of points on
                    /* top border
       numofbbe:
                   /* number of points on
                   /* bottom border
   XYCRD *bound,
                   /* pointer for boundary
                    /* data
         *tbe,
                    /* pointer for top
                    /* border elements
         *bbe:
                    /* pointer for bottom
                    /* border elements
) OBJECT:
```

where XYCRD is another data structure for a data position. Figure 2 illustrates the 'bound', 'tbe', and 'bbe'.

In order to link the local objects in the nodes globally, the local objects which have top border elements or bottom border elements must be sent to their adjacent nodes and linked one after the other for all nodes. Since this step is a kind of sequential top-down linking procedure and it involves an exhaustive comparison between every pair of 'bbe' and 'tbe', it might cause some degradation in the parallelism. To maximize the parallelism, we have proposed a parallel linking algorithm in [10]. The theoretical speed-up is log_N, where N is the number of nodes being used.

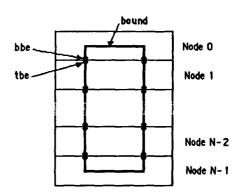


Figure 2. Boundaries (bound), bottom border elements (bbe), and top border elements (tbe).

3. Parallel Object Thinning

For load balancing, the root node collects the information of the number of objects and their sizes from all nodes. Then the root node divides the number of objects according to the number of nodes being used and the object sizes and redistributes the objects to all nodes. In the global object-linking step described in the previous section, the boundary data of the objects might be shuffled. To rearrange the data, we project the objects on a 'working plane' and perform the boundary-following step once again.

Now, we thin the objects in each node in parallel by following the object boundaries clockwise and by shrinking them in the direction perpendicular to the boundary and pointing toward the inside of the object. This procedure is repeated until the number of boundary points is not changed. Note that we find the direction for shrinking based on the eight boundary-following directions shown in Figure 3 and defined by the following equation:

$$shrink dir = (follow_dir + 2) mod 8$$
 (3)

where if shrink_dir is zero, then shrink_dir is reassigned to eight.

Figure 4 shows the result of shrinking a simple cross object after only one iteration. The starting point is the top-left position of the object, and the arrows represent the boundary-following direction which is clockwise. 'x' and '.' denote the boundary of the original object and the boundary of the shrunken object, respectively. Note that the circled points are inserted

to make the shrunken object boundary connect. The connectivity of the shrunken-object boundary is essential for the next iterations. If a boundary point is an element of a parallel line or a single line (overlapped) then the point is just copied without shrinking. The parallel line and the single line are depicted in Figure 5(a) and 5(b), respectively.

The shrinking step produces the skeletons of the objects, which are at most two-pixels wide. To make single-pixel wide skeletons, we use the Zhang and Suen algorithm [2] which preserves the connectivity of the skeletons. Note that we can remove two-pixel-wide points by following the skeleton data points instead of scanning all over the working plane. Also note that we need only one iteration, that is, two subiterations.

4. Experimental Results

Our parallel-thinning algorithm was implemented on a hypercube-topology multiprocessor computer, the Intel iPSC/2. Figure 6(a) shows a test image which contains sixteen 'H's. The size of the image is 512x512. According to the input image distribution method discussed in section 2.1, the input image was divided by the number of nodes and each sub-image was distributed to each node. Then the boundary detection for each subimage was performed in parallel. Figure 6(b) shows the result of the boundary detection from the original image. Through the parallel linking procedure, the sixteen objects were extracted and redistributed to the nodes as evenly as possible. Finally, the objects were thinned by boundary-following and shrinking. Figure 6(c) is the final result. The skeletons are singlepixel wide.

For the comparison, we also implemented Zhang and Suen's algorithm on the iPSC/2. As we discussed in the introduction, we needed to swap data between nodes at every iteration. The processing times of Zhang and Suen's algorithm as well as our algorithm according to different numbers of nodes are shown in Table 1. The data described above was used for both cases. We can see that our algorithm is much faster than Zhang and Suen's. But our algorithm has some degradation when using 32 nodes because there are only 16 objects in the image and more communication time is needed for object extraction as more nodes are used.

5. Conclusions

In most conventional thinning algorithms, the complexity of the whole procedure is $O(n^2)$. Moreover, the algorithms are undesirable for distributed memory computers. To solve these problems, we have presented a new parallel-thinning algorithm which is based on boundary following and shrinking. Also we have implemented this algorithm on the Intel iPSC/2. Theoretically, the complexity of our algorithm is $O(n^2)$. According to the experimental results, our algorithm is $O(n^2)$. According to the experimental results, our algorithm depends on the number of objects in the input image. That is, the more objects in the scene, the more efficient the algorithm. Another problem is that, so

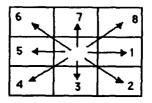


Figure 3. Eight boundary-following directions.

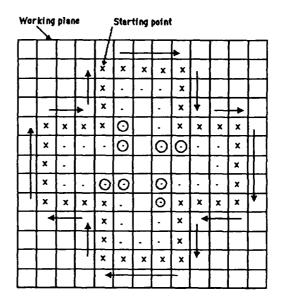


Figure 4. The result of shrinking a simple cross object after one iteration.

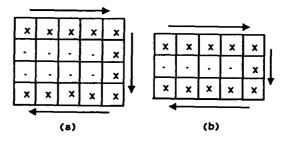
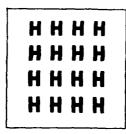


Figure 5. (a) Parallel line
(b) Single line (overlapped)

far, our algorithm works only for solid objects. But this problem can be solved by some modification of the algorithm: If an object contains some holes inside, then we detect the boundaries of the holes as well as the boundary of the object. In the shrinking procedure, we shrink the boundary of the object inward and shrink the boundaries of the holes outward.



(b)

Figure 6. (a) An origina! input image (512×512)
(b) A boundary-detected image

(c)

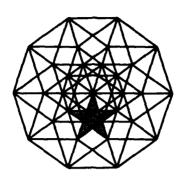
(c) A thinned image

Algorithm of nodes	4	8	16	32
Zhang and Suen	56.1	34.4	23.6	13.0
Ours	10.2	6.4	3.7	4.5

Table 1. Comparisons of the processing times

References

- [1] M. Pilar Martinez-Perez (1987), "A thinning algorithm based on contours", Computer Vision, Graphics, and Image Processing 39, pp. 187 21.
- [2] T. Y. Zhang and C. Y. S. (Mar. 1984), "A fast parallel algorithms... thinning digital patterns", Communications of the ACM 27, Num. 3, pp. 236-239.
- [3] M. E. Lu and P. S. P. Wang (Mar. 1986), "A comment on a fast parallel algorithm for thinning digital patterns", Communications of the ACM 29. Num. 3, pp. 239-242.
- the ACM 29, Num. 3, pp. 239-242.
 [4] Roland T. Chin and Hong-Khoon Wan (1987), "A one-pass thinning algorithm and its parallel implementation", Computer Vision, Graphics, and Image Processing 40, pp. 30-40.
- [5] Christopher M. Holt, Alan Stewart, Maurice Clint, and Ronald H. Perrott (Feb. 1987), "An improved parallel thinning algorithm", Communications of the ACM 30, Num. 2, pp. 156-160.
- [6] Richard W. Hall (Jan. 1989), "Fast parallel thinning algorithms: parallel speed and connectivity preservation", Communications of the ACM 32, Num. 1, pp. 124-131.
- [7] Zicheng Guo and Richard W. Hall (Mar. 1989), "Parallel thinning with two-subiteration algorithms", Communications of the ACM 32, Num. 3, pp. 359-373.
- [8] James T. Kuehn, Jeffrey A. Fessler, and Howard Jay Siegel (1985), "Parallel image thinning and vectorization on PASM", Proceedings of CVPR '85: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 368-374.
- [9] Yoshiaki Shirai (1987), Three-Dimensional Computer Vision, Springer-Verlag, New York.
- [10] Joong H. Baek and Keith A. Teague (1989),
 "Parallel object representation using straight
 lines on the hypercube multiprocessor computer",
 Proceedings of the 4th Conference on Hypercube,
 Concurrent Computers, and Applications, Vol. 2,
 pp. 987-990.



The Fifth Distributed Memory Computing Conference

3: Multi-Target Tracking

A Nonconvex Cost Optimization Approach to Tracking Multiple Targets by a Parallel Computational Network

Kenneth Rose, Eitan Gurewitz and Geoffrey Fox

Caltech Concurrent Computation Program
California Institute of Technology
Mail Stop 206-49
Pasadena, CA 91125

Abstract

The problem of tracking multiple targets in the presence of displacement noise and clutter is formulated as a nonconvex optimization problem. The form of the suggested cost function is shown to be suitable for the Graduated Non-Convexity algorithm, which can be viewed as deterministic annealing. The method is first derived for the two-dimensional (spatial/temporal) case, and then generalized to the multidimensional case. The complexity grows linearly with the number of targets. Computer simulations show the performance with crossing trajectories.

Introduction

The problem of tracking is that of estimating the trajectories of moving (point) objects given a set of noisy measurements in time. Many approaches have been suggested for tracking, some of which will be briefly mentioned here so as to clarify the relationship between them and our new method.

Two types of noise are assumed present, namely, displacement noise and clutter. The displacement noise corresponds to rrors in the location of returns with respect to the actual locations of targets. Clutter consists of noisy points which do not relate to an existing target. If only displacement noise were present, then the problem would reduce to that of curve fitting to minimize some appropriate measure. In particular, if the target dynamics could be modelled by state space equations driven by Gaussian white noise, then the Kalman filter recursive solution could be used to minimize the mean squared error criterion.

The presence of clutter, however, adds a data association aspect to the problem, i.e. which of the observed returns corresponds to the target. The Probabilistic Data Association method [2] overcomes this difficulty by consider-

ing only the most likely associations and assigning an association probability to each hypothesis. The method outputs as state estimate the corresponding average of the conditional state estimates. Another difficulty arises when dealing with multiple targets. Unlike clutter, the presence of another target produces points with a structured distribution. Thus in the case of crossing targets, these points may be assigned high association probabilities and mislead the estimator. This gave rise to the Joint Probabilistic Data Association method [3], which assigns joint association probabilities to sets of hypotheses. The complexity of this method clearly grows combinatorially. A neural network method for approximating the joint association probabilities has recently been proposed [4].

An interesting approach to tracking is by using Dynamic Programming [5]. Here the space is discretized, and a full search through all possible states is efficiently performed by exploiting special properties of the problem. The advantage of the method is that for a single target, it will always find the optimal trajectory (within the resolution of the grid). On the other hand, the ability to resolve crossing targets is determined by the resolution since two trajectories passing through the same state will be merged by the search procedure. Refining the resolution clearly affects the complexity.

Hough Transform methods have also been suggested for tracking [6]. The Hough Transform detects trajectories belonging to a specified family of parametrized curves, by a voting procedure. It is relatively insensitive to clutter, but quite sensitive to displacement noise. Much work has been devoted to reduce the complexity of the multi-dimensional Hough Transform. It can naturally be used as a track initiator for another tracking method, by detecting possible trajectories within small windows in the raw data.

In this paper a new approach is proposed. First, the tracking problem is reformulated as a non-convex optimization problem, i.e., the minimization of an appropriate energy function. The form of this energy function is then shown to be suitable for an algorithm based on the principle of Graduated Non-Convexity (GNC) which was proposed for visual reconstruction [1]. We propose a deterministic algorithm which enables avoiding local minima. In fact the energy function is replaced by a sequence of energy functions which converges to the original energy function. The sequence starts with a convex energy function and gradually introduces nonconvexity as it approaches the final energy function.

The method is first developed for the twodimensional (space/time) case, and then generainzed to deal with the n-dimensional case. Simulation results are shown to demonstrate the performance. Finally, issues of possible parallel implementation, notably in terms of cellular automata, are discussed.

The two dimensional (time/space) derivation

As stated in the introduction, the problem is made hard by its data association aspect, i.e., which point is associated with which target. In fact, if we knew the correct data association we could easily compute the optimal trajectory since the energy function would be convex. Moreover, the analytic solution could be given in terms of Green functions. The approach in this study will be to implicitly look for the set of points to associate with a target so as to minimize the energy. From such a viewpoint, if one considers all possible trajectories for a target, one should compute its energy after assigning to it the nearest returns.

The proposed energy function for twodimensional (spatial-temporal) data is

$$E = \sum_{i} \min_{j} \{ (u_{i} - d_{i}^{(j)})^{2} \} + \mu \sum_{i} (\ddot{u}_{i})^{2} + \alpha \sum_{i} \left(\frac{u_{i} - \hat{u}_{i}}{\sigma_{i}} \right)^{2} , \quad (1)$$

where u_i is the trajectory location at time i, $d_i^{(j)}$ is the j'th data point at time i, and \hat{u} is some prediction of the trajectory given past data or other external information such as other sensors etc., which may be nonuniformly weighted in

time (σ_i) . The first term of the energy function measures the trajectory's distance from the observed data. The second term penalizes nonsmooth trajectories. The third term takes into account predictions and allows adding external information.

This energy function has many local minima because of the first term which is not convex, and indeed, the first term contains the data association problem. Reconsider the first term,

$$E_1 = \sum_i g_i(u_i), \qquad (2)$$

where

$$g_i(x) = \min_j \{ (x - d_i^{(j)})^2 \}.$$
 (3)

We wish to find a *convex* approximation E^* to the energy function, and we shall do it by replacing the functions g_i by some g_i^* . The condition for convexity is that the Hessian be positive definite. The Hessian of E^* is given by

$$H_{ij}(u) = \frac{\partial^2 E^*}{\partial u_i \partial u_j}$$

$$= \frac{d^2 g_i^*}{dx^2} (u_i) \delta_{ij} + 2\mu(Q^2)_{ij} + \frac{2\alpha}{\sigma^2} \delta_{ij}$$
(4)

where δ_{ij} is the Kronecker delta and Q is the matrix given by

$$Q_{ij} = \begin{cases} 2, & \text{if } i = j; \\ -1, & \text{if } |i - j| = 1; \\ 0, & \text{otherwise.} \end{cases}$$
 (5)

Since the matrix Q^2 is positive semidefinite, then by requiring the diagonal matrix representing the first and third terms in (4) to be positive definite we ensure that so is H and therefore E^* is convex. Hence we require

$$\frac{d^2g_i^*}{dx^2} \ge -\frac{2\alpha}{\sigma_i^2} = -c_i. \tag{6}$$

The functions g_i are piecewise parabolic as illustrated in Figure 1. The best approximating functions (from below) g_i^* which satisfy (6) are obtained by fitting inverted parabolas to the boundaries as shown in Figure 1. These functions are differentiable and their derivative is continuous. Between two detected points, 2d apart, we get (assuming the origin is at the midpoint)

$$g_i^* = \begin{cases} \frac{c}{1+c}d^2 - cx^2, & \text{if } |x| < \frac{d}{1+c}; \\ (d-x)^2, & \text{otherwise.} \end{cases}$$
 (7)

Note that for $c=c_i$ we get the convex approximation we needed, while on the other hand for $c\to\infty$ we get $g_i^*\to g_i$ and therefore $E^*\to E$.

One may therefore choose c as a natural parameter for gradually introducing non-convexity into the energy function. This is not the only possibility, another choice of parameter which is closely related to multi-scale methods is currently under investigation.

The algorithm will therefore be in the following lines. Initialize $c=c_i$ so that the energy function is convex, and optimize using your favorite method (e.g. gradient descent). At each iteration increase c to introduce some nonconvexity and re-optimize. An important issue is that of where to stop the iterations. Recall that

$$g_i^*(x) \le g_i(x), \quad \forall x$$
 (8)

which implies that if the configuration u^* globally minimizes E^* , then

$$E^*(u^*) \le E(u), \qquad \forall u. \tag{9}$$

Hence, u^* is the global minimum of E if and only if

$$E(u^*) = E^*(u^*).$$
 (10)

In certain cases it turns out that the convex approximation is already a good enough approximation of the energy function (this depends on the choice of parameters) so that (10) holds and the global minimum is found. Moreover, if by choosing a careful schedule for updating c we can ensure that we are always at the global minimum of E^* , then whenever we reach a configuration which satisfies (10), we have found the global minimum of E. Note that each of the intervals over which $g_i \neq g_i^*$ is made smaller as c is increased.

Generalization to the n-dimensional space

The generalization will be given for the ndimensional spatial case and illustrated for the two dimensional spatial case. The main issue here is to produce a convex approximation to the energy function. Once we have that, we shall immediately see how to introduce non-convexity.

Again let us consider the first term of the energy function. It is a set of paraboloids centered at the data points. Over a two-dimensional space, the energy function looks like an irregular egg tray. The boundaries within which each paraboloid is defined are given by the appropriate Voronoi diagram. This is a set of hyperplanes which encloses with each data point all the points in space which are nearest

to this data point. In order to obtain the convex approximation we smooth the function over these boundaries to satisfy the second derivative requirements. Similarly to the one-dimensional case (7), the function is modified within a sleeve around the boundary hyperplanes.

The form of the approximating function at a given point will depend on the number of data points associated with it. For the case of two dimensional space, a point is associated with two data points if it is in a sleeve, and with three data points if it is in the intersection of two sleeves. In general each point may be associated with up to n+1 data points (excluding pathologies). Now suppose that we are in a zone that is associated with k+1 data points. These points are all in a k-dimensional subspace. Moreover, assuming they are "generally positioned", i.e., no (k-1)-dimensional subspace contains all of them, then they are on some k-dimensional hypersphere (which will be simply referred to as sphere).

The approximating function is defined as an inverted paraboloid over the k-dimensional space centered at the center of the bounding sphere, and an upright paraboloid in the remaining orthogonal directions.

$$g_i^*(x_1,...,x_n) = K - c \sum_{j=1}^k x_j^2 + \sum_{j=k+1}^n x_j^2$$
 (11)

where K is a constant to be determined, k+1 is the number of data points associated with $(x_1, ..., x_n)$. These data points are in fact the vertices of a hyperpolyhedron in the k-dimensional space. The intersection of the corresponding sleeves is a smaller polyhedron congruent to it whose bounding sphere has the same center (see Fig. 2 for the two-dimensional case). Let R be the radius of this hypersphere, then

$$K = cR^2 + g_i(v) \tag{12}$$

where v will stand for any of the vertices of the sleeve intersection. Note that g_i has the same value for all these vertices (equally distant from data points). Note also that for the one-dimensional spatial case we obtain (7) from (11) and (12) by substituting R = d/(1+c) which is indeed the radius of the one-dimensional bounding sphere, i.e. half the distance.

We shall omit the details here but it is not difficult to show that the approximating functions g_i^* are continuous, differentiable and their

derivative is continuous everywhere. Such an approximating function is shown in Fig. 3.

We have generalized our convex approximation to multi-dimensional spaces, and the resulting energy can still be naturally parametrized by c to introduce non-convexity.

On parallel implementation of the algorithm

In the previous sections we have constructed the sequence of energy functions which starts with a convex approximation and converges to the original energy function. However, the actual computation in the algorithm does not involve evaluating these functions everywhere. All that is required at each iteration is to evaluate the derivative with respect to each variable at the current point. As the g_i^* functions are defined by cases, the main problem is to establish the case, i.e., with how many and which data points it is associated. By geometrical considerations, it can be shown that given the current point and the nearest data point, all that is required is to search a certain window for additional data points. The window is defined as the difference of two hyperballs B-b, where b is a ball centered at the current point and whose radius is the distance to the nearest data point

$$r = |x - d^{(j)}|.$$
 (13)

The larger ball B is the interior of a sphere passing through the data point, whose center is on the line connecting the two points, and whose radius is

$$R = \frac{1+c}{c}r. (14)$$

This is illustrated in Fig. 4. The data points found in the crescent B - b will determine the form of g_i^* .

Let us reconsider the energy function given in (1). As there is no interaction between tracks, the complexity grows linearly with the number of tracks. In fact, all trajectories can be computed in parallel. We shall next discuss possible parallelization of the computation of a single trajectory. The second observation to make is that trajectories are temporally but not spatially discretized.

The fact that the trajectories are not discretized in space allows avoiding apriori limitations on resolving crossing targets. The discretization of space into a large number of mutually exclusive states is typical for neural-network

formulations and dynamic programming methods.

On the other hand, the discretization in time which is assumed to be property of the input, enables a parallel implementation. This can be done by a network of processors, each in charge of a given time slice. The only inter-processor communication is within small neighborhoods, through the smoothness term of the energy function which contains a temporal derivative. It is therefore natural to visualize such a system in terms of cellular automata. For a given time window, each cell processes one time slice data while incorporating into the processing the output of its defined neighbors.

In order to eliminate the need to transfer input data between processors as the time window slides, a cyclic index rotation is used. The processors are connected in a circle, and the connection is severed between the last and the first time slices in the window. As the window slides by one time unit, all indices are rotated so that each processor still deals with the same input data, but advances within the window. The processor which was last now becomes first and receives fresh input. Note that the disconnected branch is also rotated to be between the current last and first time slice in the window.

Simulation

A simulated example is shown in Fig. 5. There is one spatial dimension and one temporal dimension. Five crossing targets are detected in the presence of clutter and displacement noise. The targets were generated by specifying initial positions and velocities, and applying small acceleration noise to them at each time unit.

Summary

A nonconvex cost optimization approach is suggested for multitarget tracking in the presence of displacement noise and clutter. The method is based on deriving a convex approximation to the energy function and gradually introducing nonconvexity. By this procedure we start with the global minimum of the approximated energy function, and perform "tracking" of the global minimum while varying the nonconvexity parameter. In this respect the method can be viewed as deterministic annealing. The convex approximation was derived for the two-dimensional (spatial/temporal) case and then generalized for multi-dimensional cases. The

computations can be performed in parallel per track and per time slice. A simulated example is presented to demonstrate the performance of the method.

References

- [1] Blake, A., and A. Zisserman, Visual Reconstruction. Cambridge, MA: MIT Press, 1987.
- [2] Bar-Shalom, Y., and E. Tse, "Tracking in a cluttered environment with probabilistic data association," Automatica, Vol. 11, pp. 451-460, Sept. 1975.
- [3] Fortmann, T. E., Y. Bar-Shalom and M. Scheffe, "Sonar tracking of multiple targets

- using joint probabilistic data association," IEEE Journal of oceanic Eng., July 1983.
- [4] Sengupta, D., and R. A. Iltis, "Neural solution to the multitarget tracking data association problem," IEEE Trans. on Aeros. and Electr. Systems, vol. 25, pp. 96-108, Jan. 1988.
- [5] Barniv, Y., "Dynamic programming solution for detecting dim moving targets," IEEE Trans. on Aeros. and Elect. Systems, vol. 21, pp. 144-156, Jan. 1985.
- [6] Krishnapuram, R., and D. P. Casasent, "Hough transform detection of 3-D curves and target trajectories," Applied Optics, vol. 28, pp. 3479-3486, 1989.

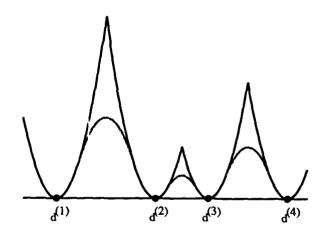
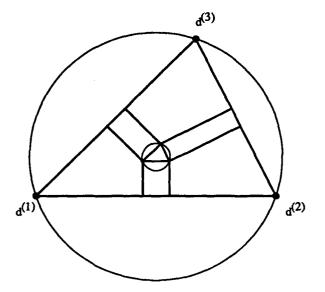


Figure 1. The function g and its approximation g^*



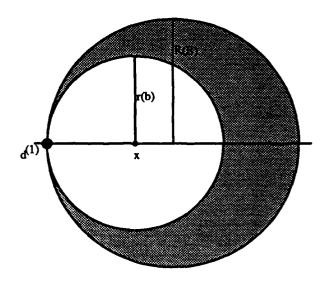


Figure 2. Partition of the domain of g^* according to its definition.

Figure 4. The search window is given by the shaded region B - b.

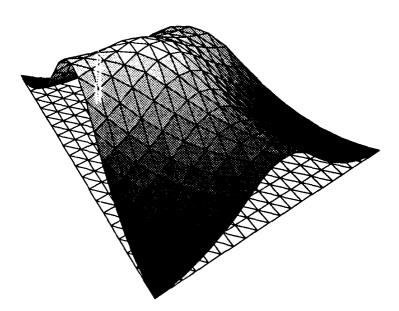


Figure 3. The approximation g^* between three data points

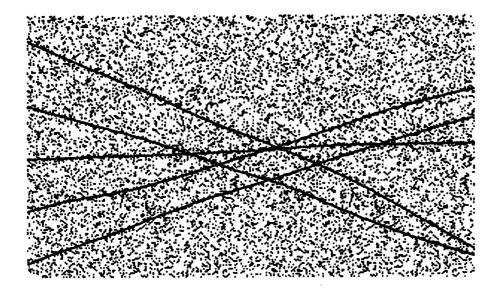


Figure 5(a). The original trajectories plus clutter.

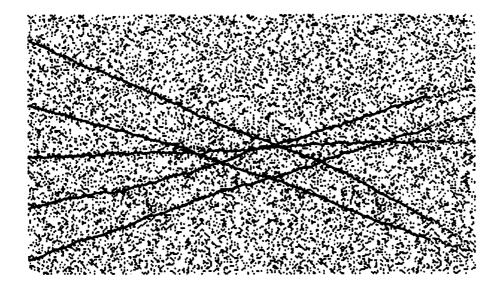


Figure 5(b). The computed trajectories.

Concurrent Multi-Target Tracking

T. D. Gottschalk California Institute of Technology Pasadena, CA 91125

1 Overview

Simulation89 is an emulation of various SDI tasks (tracking, engagement management and 'look ahead') developed for the U. S. Air Force. The simulation presently deals with the boost, post-boost and early midcourse phases of a 'mass raid' scenario, and is designed to process scenarios with a few thousand targets. The simulation is run on the Mark-III hypercube, with individual tasks performed on subcubes of the full hypercube. In general, the computations within individual subcubes are done in a synchronous manner (i.e., CrOS), while communications between tasks/subcubes are done asynchronously.

The nominal task for the tracking module is to provide state information on individual targets, given 2D line of sight data from various sensors at regular time intervals. This task is complicated by means of a few relevant additional requirements:

- 1. In the initial boost phase, the trajectories of individual targets are not fully known.
- 2. The overall system must scale in such a way that increases in the size of the underlying scenario are accommodated by (proportional) increases in the size of the tracking sub-cube.
- 3. The tracker must meet 'real time' requirements.

The first requirement in fact dictates the gross overall structure of the tracking package, as illustrated in Fig.(1). A single tracking system is formed from two elementary 2D tracking subsystems. Each 2D tracking sub-system processes individual data from its own associated sensor, forming lists of plausible mono tracks through these data sets. These 2D tracks are then shared between the two 2D sub-systems, and a single set of 3D tracks is formed.

The tracking models used for the 2D and 3D subsystems are quite different. According to the first requirement listed above, it must be assumed that the data from a single sensor are insufficient to resolve all Track→Hit ambiguities. As a consequence, the 2D systems use a Multiple Hypothesis formalism in which

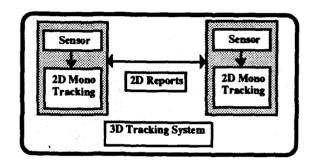


Figure 1: Schematic Tracker Organization

many candidate tracks through a single sensor datum are allowed. Such a model is subject to exponential explosions of the overall track file. This fundamental difficulty is resolved by a number of rules for pruning the size of the overall track file. In particular

- 1. Two tracks ending on a given datum are said to be equivalent if they share the same 2D data over the last four scans.
- 2. The number of inequivalent tracks per datum is limited by a cutoff parameter.
- 3. The total number of 2D tracks is also limited by a global cutoff.

If two tracks in the system are found to be equivalent according to point 1, one of the tracks is simply deleted. As is discussed below, the task of identifying equivalent tracks in the distributed 2D track file dictates the maner in which the 2D tracking problem is decomposed for concurrent execution.

Unlike the 2D tracking system, the 3D tracker in Fig.(1) maintains (at most) one track per sensor data point, representing the best global interpretation of tracks through the data (this single 'best guess' answer is the output of the tracker expected by the other elements of Sim89). In place of the Multiple-Hypothesis model used for 2D tracking, the

3D tracker is based on *Optimal Associations*. These optimal associations in fact come in two distinct forms:

- 1. For track extensions, the predicted data positions of individual 3D tracks are associated with actual data from the two sensing subsystems of Fig.(1).
- 2. For 3D track initiations, 2D tracks form the two subsystems of Fig.(1) are associated according to values of projections onto an association reference axis (so-called 'Hinge Angle' associations).

The adoption of an Optimal Association formalism essentially trivializes the concurrent decomposition of the 3D tracker: the 3D tracks are distributed among the nodes of the subcube in such a way that the number of tracks per node is constant. The challenge of concurrent 3D tracking comes entirely in performing the two types of optimal associations.

A general concurrent algorithm for optimal associations is described in Ref.[1]. However, the resource requirements for the general optimal association problem ($\Delta T \propto N^3$ for $N \times N$ association problems) is such that a straightforward use of the general association formalism is completely inappropriate. Instead, the concurrent association algorithm proceeds as follows:

- Each node computes a list of associations keys (i.e., projections onto an appropriate reference axis), for all items in its local track list.
- 2. The distributed lists of association keys are globally sorted.
- The sorted lists are divided into a number of subblocks, determined by appropriately large gaps in the lists of keys.
- 4. The sub-blocks are assigned to individual nodes and the assignment problems for sub-blocks are solved using a modified 'sparse' formalism of the general assignment problem.

This procedure is efficient as long as the number of separate sub-blocks found in the third step is large compared to the number of nodes in the tracking subcubes (which is, empirically, almost always the case for the Sim89 problem).

In addition to the central tasks of Track↔Hit and Track↔Track associations, the 3D tracker also evaluates trajectory fits for all 3D tracks in the system. Unlike the predecessors to Sim89, these trajectory fits are not essential elements of tracking per se. All tracking is done using kinematic system models. The trajectory parameterizations are added to the tracking task

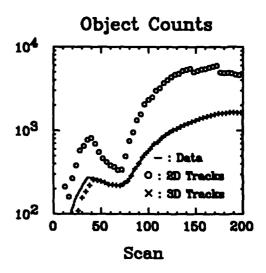


Figure 2: Data Set and Track File Sizes Versus Scan

for purposes of communications: The 3D track file structures according to the kinematic model are huge (more than 100 floating point numbers per track), and the model-dependent parameterizations greatly reduce the sizes of track file messages passed between subcubes of the full Sim89 simulation. The concurrent estimation of track parameters is again trivial, with each node independently performing this task for its own subset of the global track file.

2 Some Sample Results

This section briefly examines some typical results of the Sim89 tracker for a standard input set. The threat scenario involves 200 primary targets, each of which ultimately spawns 10 daughter objects (RV's). The targets are launched from six separated launch sites over a two minute time window. The primary threat is preceded by a simultaneous launch of sixty secondary targets (ASAT's).

Sizes of the data sets and 2D and 3D track files are plotted versus scan number in Fig.(2). The peaks near scan 40 are due to interception of the ASAT's, while the prolonged increase in object counts after scan 80 is due to gradual deployment of RV's from the surviving primary targets. As expected, the number of 2D tracks greatly exceeds the actual number of targets. The 'kinks' in the 2D track counts for large scan number are the result of the automatic reductions in tracks/datum cutoffs mentioned in Section 1.

The number of 3D tracks is very close to the actual number of targets. The histogram in Fig.(3) shows

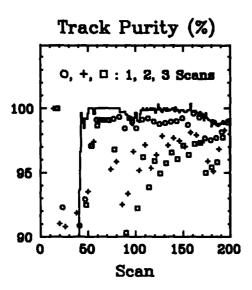


Figure 3: Track File Completeness, Purity Versus Scan

the percentage of targets in track,

$$P[In Track] \equiv N[3D Tracks]/N[Data]$$

versus scan number. Once the primary targets are into second stage (about scan 50), the percentage in track is excellent. Also shown in Fig.(3) are the fractions of pure tracks

$$P_j \equiv N[j\text{-Scans Correct}]/N[3D \text{ Tracks}]$$

where the numerator is the number of 3D tracks which (correctly) incorporate data from a single underlying target through the past j scans.

The mild degradations in both percentage in track and j-Scan correct tracks between scans 150 and 200 are due to the successes of the engagement management component of Sim89 in intercepting the targets. The disappearence of expected targets causes some 'confusion' for Track→Hit associations on subsequent tracking scans.

The CPU times for various components of 3D tracking are plotted versus scan number in Fig.(4). Most of the CPU resources are spent in the evaluation of Track \leftrightarrow Hit associations. With the exception of the 'confused' scans with disapperaing tracks, the CPU requirements for tracking generally scale as $\Delta T \propto Nlog N$ for N active targets.

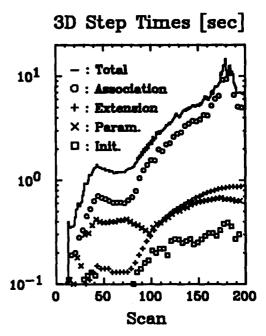


Figure 4: CPU Step Times For 3D Tracking

3 Concurrent Aspects

The task of multi-target tracking is well-suited for concurrent execution, with most of the 'tracking' per se done by way of CPU-intensive operations involving individual track → data pairs (the filter update of a single 3D track involes more than one thousand floating point operations: an ideal use of the WEITEK coprocessor). In the entire tracking program (more than 35000 lines of code), there are really only three general concurrent operations/aspects,

- 1. Global collection of data across the subcube.
- 2. Distributed sorting.
- 3. Track file redistributions.

with each of these tasks occuring in a variety of guises. The sorting task is done using the basic algorithm of Ref.[2], with a trivial but important modification to allow empty local sublists (empty track files on some nodes occur during the first few scans of the tracking task).

The global data collections are all done using a simple loop on communication channels:

- Set Global Value To Local Value
- Loop On Communication Channels
 - Exchange Values Across Channel

- Update Global Value Using Input

• End Of Channel Loop

This simple paradigm is used throughout the code for a variety of purposes, such as assessment of global status (the 'Update' task is a logical and of individual status flags), determining global file sizes ('Update' is simple addition) or assessing global Track \leftrightarrow Data assignments ('Update' is a slightly more complicated merging of track assignment arrays generated on individual nodes).

For the 3D tracking task, concurrent efficiency requires only that the number of tracks per node be approximately constant. Accordingly, track file redistribution for 3D tracking is done using a simple variant of the channel loop model:

- Loop On Communication Channels
 - Exchange Track File Sizes
 - Set $\delta \equiv (N_{\text{HERE}} N_{\text{THERE}})/2$
 - If $\delta > 0$, Send δ Items Over Channel.
 - If $\delta < 0$, Receive δ Items.
- End Of Loop On Channels.

After the exchanges across a given channel, the number of items on each half of the subcube with respect to that channel is (approximately) the same, and subsequent loops on other channels do not modify this equality. At the end of the channel loop, the tracks are equally divided across all channels - meaning that the number of tracks per node must be approximately the same.

The only aspect of the full tracking program which involves concurrent 'subtleties' is the redistribution of the the global 2D track files. Wasteful cube-wide searches for equivalent tracks (same sensor data over the last four scans) can be avoided if the assignment of tracks to nodes is done according to a single essential requirement.

At the start of each 2D tracking scan, all tracks ending on a given sensor datum are to be assigned to a *single* node.

If this condition is satisfied, then searches for equivalent tracks need only be done locally. The requirement is enforced as follows:

- 1. Assign each datum of the current data set to a specific node.
- 2. Transfer all tracks in the system to that node which 'owns' the data point for the last scan included in the track.

This redistribution is in fact done as the last step in 2D processing at each scan, so that the next scan begins with the basic track distribution requirement satisfied.

Once data points (hence tracks) have been assigned to individual nodes, the actual redistribution of the tracks is a straightforward application of the basic Crystal_Router formalism of Ref.[3]. The calculation of destinations for individual data is done using the following simple set of rules:

- 1. Each datum is assigned a Weight, taken to be the total number of tracks in the system wich end at that datum.
- Data are assigned to nodes such that the total Weight per node is approximately constant.
- 3. If, prior to the redistribution, a particular node already contains more than half of the total weight of an individual datum, then the datum is assigned to that node provided that such an assignment does not violate the total node weight restrictions of point 2.
- Unassigned data (i.e., data without tracks) are assigned to nodes in a simple 'Card Dealing' fashion.

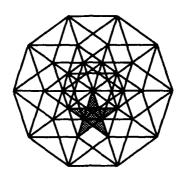
These rules are easily implemented by means of a few simple channel loops of the form discussed above.

4 Conclusion

The hard part of 'Concurrent Tracking' is the tracking iteslf, not the concurrency.

References

- 1. T. D. Gottschalk, 'Concurrent Implementation of Munkres Algorithm', there proceedings.
- 2. G. C. Fox et al., Solving Problems on Concurrent Processors, Englewood Cliffs, NJ: Prentice Hall(1988), Chapter 18.
- 3. Chapter 22 of Ref.[2].



The Fifth Distributed Memory Computing Conference

4: Simulation of Systems and Discrete Events

Parallel Discrete Event Simulation Using Synchronized Event Schedulers

William L. Bain

Block Island Technologies 15455 NW Greenbrier Parkway, Suite 210 Beaverton, Oregon 97006

Abstract

This paper describes a new algorithm for the synchronization of a class of parallel discrete event simulations on distributed memory, parallel computers. Unlike previous algorithms which synchronize on a per process basis, this algorithm synchronizes on a per processor basis. The algorithm allows full generality in the simulation model by allowing dynamic process creation and destruction and full inter-process interconnections, and it is shown to be deadlock and livelock free. It has been used to simulate very large parallel computer architectures.

This algorithm has been implemented on the Intel iPSC/2 parallel computer system, and its performance has been measured. The algorithm achieves a time overhead of O(log2 N) for binary hypercube systems with N processors and O(D) in general, where D is the diameter of the parallel system. In order to obtain good overall parallel speedup, the algorithm requires that the simulation generate at least O(N) events at each simulated time.

Introduction

As discrete event simulations increase in size and complexity, it becomes advantageous to execute them on parallel architectures. Parallel architectures offer the processing power to execute multiple processes concurrently, thus speeding up the simulation. In addition, they provide sufficient physical memory to hold large simulation models without suffering the

delays required to access backing store in virtual memory systems.

In order to avoid bottlenecks to parallel speedup [1], parallel synchronization algorithms have been developed. These algorithms distribute the event scheduling algorithm among parallel system's processes, which eliminates the hot spot in memory access patterns arising from the use of a centralized event scheduler, a problem that occurs in both shared and distributed memory parallel computers. Two principal classes of parallel synchronization algorithms have emerged. The conservative approach [1],[2],[3] constrain the processes to handle incoming events in strict time order. These algorithms take steps to ensure time order before processing events. optimistic approach [4] allows events to be handled in their order of arrival and provides rollback and recovery mechanisms to handle events processed out of order in time.

Algorithms developed for both approaches distribute the synchronization algorithm on a per process basis, thereby allocating one logical clock and event scheduler to each process in the simulation model. so doing, these algorithms incur a total time and memory overhead to advance all process clocks one time interval that is at least proportional to the number of processes, P, in the system; denote this overhead the synchronization overhead. Conservative algorithms may suffer time than overheads much greater Ρ. depending on the connectivity of the inter-process communication graph.

occurs because each process must determine the state of all processes that may send it an event message before it can safely advance its time. Hence, in a system with N parallel processors, these algorithms incur a time overhead of at least O(P/N) to advance all P clocks one time interval each.

Because of the practical need to maximize the number of processes executed by each in current processor systems, synchronization time overhead typically becomes at least O(N). With today's parallel computer technology, important to maximize the problem size per processor in order to amortize the interprocessor communication delays and maximize the observed speedup In systems with several megabytes of memory per processor, such as the Intel iPSC/2 system, and typical process memory requirements of 2-6K bytes, one can execute multiple thousands of processes per processor (i.e., P/N > 1000). Thus, for systems with up to a thousand processors (i.e., N <= 1000) these simulations will encounter a time overhead of at least O(N).

This paper describes a new parallel synchronization algorithm using multiple synchronized event schedulers, one per processing node of the system. The use of multiple event schedulers was described for the Yaddes simulation environment However, the Yaddes mechanism uses [7]. centralized synchronizer and upon knowledge of the interconnection between logical processes. It also requires that N * (N-1) messages be sent per time step, which results in an overhead of at least O(N) to advance all P clocks by one In contrast, the algorithm time step. described here has been demonstrated to synchronization overhead have a O(log₂ N) on hypercube based parallel It will in general have an overhead of O(D), where D is the diameter of the parallel system. The diameter is defined here as the maximum path length between the root and leaf nodes within a processors; it spanning tree of all represents the number of time steps

required to either broadcast from one processor or aggregate data from all processors.

The next section describes the algorithm. The following section provides recent performance measurements of the algorithm's time overnead. This algorithm has been implemented as part of the Interwork IITM software package [8] on the Intel iPSC/2 parallel computer system [9]. It has been used to simulate very large parallel computer architectures [10].

The Synchronization Algorithm

The synchronization algorithm described here uses one event scheduler per processor of the system. All of the processes residing on a given processor share the same event scheduler, which consists of a logical clock and a time ordered queue of pending events. These event schedulers are synchronized in a conservative manner so that no event scheduler may advance its clock until it can be sure that no other event scheduler will send one of its processes a lower time event.

In order to allow full generality to the simulation model, the event schedulers are fully connected. That is, processes within the simulation can send messages to arbitrary other processes. This eliminates the need to describe a static inter-process communication graph, in as conservative algorithms, and it allows processes to be dynamically created and These characteristics greatly destroyed. simplify the description of complex applications. Full interconnection event schedulers also simplifies the partitioning of processes to processors. process can be allocated to any processor, and it may send an event message to any other process in the system.

The use of full connectivity between event schedulers requires that all event schedulers be synchronized to the same value of global system time. The algorithm synchronizes the event schedulers using multiple, distributed

spanning trees, one rooted at each event scheduler, to collect and distribute clock times in parallel and with minimum total delay. The event schedulers initialize their clocks to time zero and execute their the simulation portion of processes until they complete all events at After each event scheduler this time. becomes quiescent, it exchanges its next event time, i.e., the lowest time at which it can next process an event or infinity if no events are enqueued, with each of its in Each scheduler neighbors turn. determines the minimum of all collected In this and its own next event times. all event schedulers determine in parallel the globally minimum next event time. The event schedulers advance their clocks accordingly and execute all events at the The algorithm repeats these new time. actions for each event time.

By synchronizing all event schedulers to the same global time, the algorithm ensures that no process can send a lower time event to another process in the This guarantees the correctness system. of the algorithm. The algorithm is deadlock free because the system is guaranteed to advance to a new time after all event schedulers exhaust their events current time, which the It is also livelock free eventually must do. sequence of globally because the minimum next event times is strictly increasing. A next event time of infinity can only be reached when there are no events in the system left to process.

The algorithm's relative simplicity makes it straightforward to implement. However, implementations of this algorithm must handle the race condition that occurs when a process sends an event message to a process at another event scheduler just before the first process's event scheduler becomes quiescent. If the second event scheduler has already become quiescent and reported its next event time, its neighbor(s) could determine an erroneous globally minimum next event time while an active event message at the current time is in transit. The sending process

must report to its event scheduler that the system is still active even though its local event queue has become empty.

Performance of the Algorithm

The use of a spanning tree to collect next event times incurs a time overhead of O(D), where D is the diameter of the system. The broadcast of the globally lowest next event time requires a second O(D) time. Thus the total time overhead to advance the clock is O(D). For binary hypercube systems, such as the Intel iPSC/2 system,

$D = log_2 N$.

The actual performance of the algorithm was measured on an Intel iPSC/2 system with from one to 32 processors. Figure 1 shows the average time to advance the one clock interval versus the of processors for number an implementation of the algorithm. In measure order to only the synchronization overhead, the event schedulers handled no actual events per time step. The graph shows that the time grows logarithmically with the number of processors. By scaling the simulation model to maintain a constant processor load as processors are added, the algorithm thus becomes more efficient for larger systems and larger simulation models. The algorithm's time overhead remains constant as more memory and thus more processes are be added for each processor, and its fraction of the total simulation time smaller. Unlike previous becomes conservative and optimistic approaches, algorithm's performance is largely independent of the interprocess communication patterns. A dependence may occur in handling the previously cited race condition, whose frequency depends on the communication patterns, load balance, and system delays.

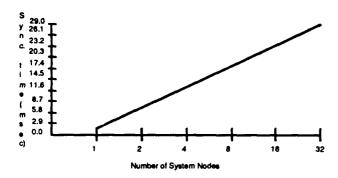


Figure 1: Time synchronization overhead per time step versus number of processors

This synchronization algorithm yields best performance for a limited class of discrete event simulation models. to obtain good overall parallel speedup, the algorithm requires that the simulation generate at least O(N) events at each simulated time and that these events be well load balanced across the processing nodes. Symmetric, discrete time simulations. such the simulation distributed memory architectures or VLSI circuits, are well suited to this scheduling mechanism. Simulations which do not meet these criteria may achieve better speedup overall using previously described schemes, despite their higher synchronization overhead.

Summary

This paper has described a new algorithm for the synchronization of parallel discrete event simulations on distributed memory, parallel computers. Unlike previous algorithms which synchronize on a per process basis, this algorithm synchronizes on a per processor basis. accomplished is bу grouping processes within one event scheduler per processor and then synchronizing the schedulers multiple, event using distributed spanning trees. The algorithm allows full generality in the simulation allowing dynamic model bу creation and destruction and full interinterconnections, and it was shown to be deadlock and livelock free.

has been used to simulate very large parallel computer architectures.

This algorithm has been implemented on the Intel iPSC/2 parallel computer system, and its performance has been measured. The algorithm achieves a time overhead of O(log2 N) for binary hypercube systems with N processors and O(D) in general, where D is the diameter of the parallel system.

In order to obtain good overall parallel speedup, this synchronization algorithm requires that the simulation generate at least O(N) events at each time step and that these events be well load balanced across the processing nodes. Symmetric, discrete time simulations, such the simulation of distributed memory architectures or VLSI circuits, are well suited to this scheduling mechanism. Future investigations will assess the applicability of this algorithm to continuous time simulations, such as the simulation of air traffic [11]. In these simulations, it may be possible to quantize the simulation clock without adversely affecting the accuracy of the results.

References

- [1] Misra, J. 1986. Distributed Discrete-Event Simulation. Computing Surveys 18, no. 1 (March): pp. 39-65.
- [2] Bain, W. L. and Scott, D. S. 1988(b). An Algorithm for Time Synchronization in Distributed Discrete Event Simulation. In *Distributed Simulation* 1988 (San Diego, Feb. 3-5). The Society for Computer Simulation, San Diego, pp. 30-33.
- [3] Reynolds, P. 1982. A Shared Resource Algorithm for Distributed Simulation. In Proceedings of the 9th International Symposium on Computer Architecture (Austin, TX, Apr. 26-29). IEEE, New York, pp. 259-266.
- [4] Jefferson, D. R. 1985. Virtual Time. ACM Trans. Prog. Lang. Syst. 7, no. 3 (July); pp. 404-425.

- [5] Moler, C. 1987. A Closer Look at Amdahl's Law. Technical Note TN-02-0587. Intel Scientific Computers, Beaverton, Oregon.
- [6] Gustafson, J. L. 1988. Reevaluating Amdahl's Law. Communication
- [7] Preiss, B. 1989. The Yaddes Distributed Discrete Event Simulation Specification on Language and Execution Environments. In Distributed Simulation (Tampa, March 28-31). The Society for Computer Simulation, San Diego, pp. 139-144.
- [8] Block Island Technologies 1988.

 Interwork II™ Concurrent

 Programming Toolkit, Reference

 Manual, Portland, Oregon.
- [9] Intel Scientific Computers 1988. iPSC®/2 Technical Description, Beaverton, Oregon.
- [10] Bain, W. and Arshi, S. 1988. Hypersim: A Hypercube Simulator for Parallel Systems Performance Modeling. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (Pasadena, Ca., Jan 19-20). ACM, Los Angeles.
- [11] Bain, William L. 1990. Air Traffic Simulation: An Object Oriented Discrete Event Simulation on the Intel iPSC/2 System, Fifth Conference on Distributed Memory Concurrent Computers (Charleston, Apr. 9-12).

Air Traffic Simulation: An Object Oriented, Discrete Event Simulation on the Intel iPSC/2 Parallel System

William L. Bain

Intel Scientific Computers 15201 NW Greenbrier Parkway Beaverton, Oregon 97006

Abstract

A discrete event simulation model of air traffic flow within the United States has been written and executed on the Intel iPSC®/2 parallel system. The simulation program was written in an object oriented IIIM manner using the Interwork Concurrent Programming Toolkit. This demonstrates how object simulation oriented programming can simplify the design of complex simulations and can simplify the effort to distribute balance the processing load on distributed memory, parallel architectures, such as the iPSC/2. It also demonstrates the capacity of these architectures to solve very large simulation problems.

Introduction

Discrete event simulation techniques can be used to model large physical systems and thereby predict aspects of their An important example of a behavior. large discrete event simulation is the modeling of the air traffic flow within the United States. Over three thousand commercial flights per hour cross the skies, and their motions must be carefully controlled to avoid congestion. Modeling this complex system allows planners to minimize delays, maximize use of available resources (such as airways, runways, and fuel), and anticipate the effects of weather and mechanical problems.

Distributed memory, parallel systems, such as the Intel iPSC/2 parallel system [1] provide an excellent hardware platform for running large discrete event

simulations. They have sufficient primary memory to hold very large simulation models; this avoids the delays encountered in paging simulation data to and from backing store. They also have the parallel processing power to exploit the inherent parallelism of the simulation For example, the many aircraft models. within an air traffic simulation independently progress along their flight paths in parallel. This provides the opportunity to speedup the execution of the simulation.

Object oriented programming techniques offer advantages in describing discrete simulation models. techniques foster the construction modular programs by associating logically related data with the procedures that This allows the manipulate the data. programmer to build new data types that extend the set of types provided by the compiler. In a simulation model, the entities are conveniently simulated object described as types encapsulate their various characteristics. In the air traffic simulation, the aircraft and airports are modeled as separate object types. Instances of these types are dynamically created (and destroyed) during the course of a simulation to represent an actual physical system. This modular approach simplifies simulation's structure.

Object oriented techniques also simplify the implementation of large discrete event simulation models on parallel systems. The modular decomposition of the simulation data provides a basis for partitioning these data among the system's processing nodes. The name space by which objects are identified and accessed provides a logical basis for communication between objects that is independent of the locations within the physical system. This maintains the simple view of the simulation program as a collection of communicating entities. It also provides a for transparently relocating the objects among the nodes (for example, to improve the load balance) without disrupting the programmer's view.

The remainder of this paper describes how the air traffic simulation was modeled on the Intel iPSC/2 system using object oriented techniques. The next section describes the object oriented simulation model. The following section describes the implementation of this model on the iPSC/2 system.

The Air Traffic Simulation Model

The air traffic simulation program models the motion of aircraft between source and destination airports through intervening sectors of controlled air space. The simulation model uses three main object types:

airplane, which models the position, velocity, and other characteristics of aircraft.

airport, which models the location, characteristics (such as runway headings and lengths) and access to air traffic control for takeoff and landing, and

sector, which models a portion of the airspace and the data required to manage the aircraft flying therein.

Instances of these object types are created to model their physical counterparts. In addition, three process types are used in the air traffic model:

pilot, which commands the actions of a single aircraft throughout a flight,

air traffic controller, which controls access to airport runways for takeoff and landing and provides separation between aircraft in the airspace sectors, and

dispatcher, which schedules new flights for departure at each airport.

Following the Hoare process-monitor model [2] for communicating processes, instances of these process types communicate and synchronize their actions by accessing instances of the airplane, airport, and sector object types. The latter three types serve as monitors in this communication model. Figure 1 shows relationships between the access process and monitor objects; a directed arc in the graph from a process type to a monitor type indicates that an instance of the process type accesses an instance of the monitor type.

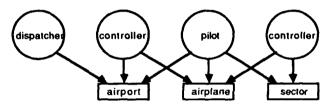


Figure 1: Access Relationships Between Simulation Objects

The use of a process-monitor model clarifies the communication relationships between the simulation objects naturally models the physical system. example, a pilot requests permission to land at an airport by accessing the airport object, which relays the request to an air traffic controller at the airport. approach models a pilot's targeting communication to an airport instead of to a particular controller at the airport. easily allows the simulation accommodate multiple air traffic controllers at the airport. Other simulation techniques (e.g., Misra [3], Time Warp [4]) model all simulated entities reactive objects which incoming messages and invoke the appropriate procedures, which turn generate messages for other objects. The communication between processes

monitors can be cast into this form. However, the use of processes appears to more clearly express the temporal behavior of active simulation entities (e.g., pilots) by collecting their sequence of actions into one thread of execution.

The initiation of a simulated flight is accomplished by dynamically creating and invoking their objects interface procedures. After a random interval and following a closed queueing model, the dispatcher process at an airport creates an airpiane object and a pilot process to control it. The dispatcher passes parameters to the pilot process identifying the airplane object and the destination. The pilot process computes the course to the destination and then requests takeoff permission bу invoking procedure of the local Request_ctlr() The airport object records airport object. the request for the controller, and the pilot process blocks awaiting a reply. airport's air traffic controller process obtains a request by invoking Deq_request() procedure of the airport After a simulated delay, it grants object. the request bу invoking the Reply_to_plane() procedure of the requesting pilot's airplane object. The pilot process unblocks and directs the airplane toward the destination airport by invoking the Change velocity() procedure of its airplane object.

The pilot process simulates a flight by sleeping for the simulated time required to reach the destination airport. (Note that in other simulation models, the pilot process could equivalently send itself a message at the future time of arrival.) This is accomplished by invoking the Sleep_flight() procedure on the local sector object, i.e., the sector object in area the departing airport is located. This procedure causes the pilot process to sleep until it reaches the destination. At this time, the pilot process permission to land at destination airport; the sequence of object invocations follows the sequence used to Takeoff and landing delays are measured and averaged for all airports.

The Sleep flight() procedure encapsulates the actions required to move the aircraft from sector to sector of the airspace until it reaches its destination. It returns control to the pilot process at an earlier time only if the aircraft comes into conflict with another aircraft and must change its course. This design allows the pilot process to simulate the actions of a real pilot in managing changes of course, while making the handoffs between airspace sectors transparent simplicity.

The sector objects manage the progress of flights across the airspace. Each sector object represents a particular portion of the air space and has an associated controller process. Pilots enter their aircraft into the local sector by executing procedure. the Sleep_flight() This procedure enqueues the flight examination by the sector controller and blocks the calling pilot process. The sector controller process repeatedly dequeues requests in its sector invoking the Get_next_request() procedure on its sector object. controller records the flight in its list of aircraft operating in the local sector. also computes the earliest time at which it will either depart the sector or come into conflict with another aircraft within the sector. The controller then unblocks the pilot process, which sleeps until this time. If the process awakens due to exiting the procedures sector, it invokes sector required to remove it from the current sector and enqueue it into the next sector along its path. If it awakens due to a traffic conflict, control returns to the pilot process to take the necessary actions. (Pilots do not take evasive action in the implementation current o f simulation.)

The use of a sector controller models the actions of its physical counterpart. It also avoids the potential deadlock that can arise if multiple pilot processes access airplane objects in order to detect conflicts and atomically update the airplane objects' states.

Implementation on the Intel iPSC/2 System

The air traffic simulation was implemented on the Intel iPSC/2 system using the Interwork H Concurrent Programming Toolkit [5]. This C language toolkit provides a global object name space spanning the system's nodes in which objects can be dynamically created. destroyed, and accessed [6]. This object name space was used to create instances of the simulation object types described above (i.e., airplane, airport, and sector). Interwork II provides a built in object type for lightweight processes; this was used to create the simulation processes dispatchers, (i.e., the pilots, and controllers) as instances of the lightweight process type.

The use of a global object name space greatly simplified the implementation of this simulation on a distributed memory, parallel system. The simulation objects can be directly referenced to invoke their interface procedures by using their global For example, a pilot process can request permission to land at an airport by knowing only the airport's name. Without a global name space, the pilot would need to know the processing node on which the airport object resides and would have to send a message to communicate with a remote airport. Interwork II's global name space transparently locates objects within the system and generates messages as necessary to invoke their procedures on remote nodes. In addition, the airplane objects transparently migrate from node to node as the sector controllers access their contents. This allows the controllers to atomically update the states of two airplanes (to re-vector them in case of a conflict) without regard to the nodes on which the airplanes reside.

The use of lightweight processes allows thousands of processes to be created, which is required for large simulations with thousands of flights and hundreds of airports. The use of the global name space to access the lightweight processes allows

them to be transparently referenced on remote nodes. For example, a sector controller can unblock a pilot process without knowing on which node the pilot process is located.

The parallel system's distributed memory architecture requires that the simulation objects be partitioned among memories of the processing nodes. use of object oriented techniques simplifies this somewhat by encapsulating logically related data into separate object To maximize the load balance of the simulation, the objects are distributed across the nodes according to their position within the airspace. Thus. parallelism in the simulation is achieved using a domain decomposition, where the domain is the simulated airspace.

In order to avoid manually placing the objects on the nodes, an Interwork II indexed object is used to represent the An indexed object is a collection of related objects, which are referenced within the collection using dimensional index [7]. Interwork automatically distributes the component objects across the nodes so as to best balance the number on each node and minimize communication delays between neighboring objects. In this simulation, an indexed object is used to represent the airspace, and each component object represents a sector of the airspace. manner, the objects sector аге partitioned transparently across the In addition. processing nodes. associating the airport and dispatcher objects with their corresponding sector objects, these other objects are transparently the partitioned across method statically This nodes. balances the simulation load within the parallel system, which works well when the airports and resulting air traffic are evenly distributed across the air space. relationship between the sector objects and their associated airport objects depicted figure in Future implementations need to dynamically remap sectors to nodes to provide dynamic

load balancing for irregular simulation loads.

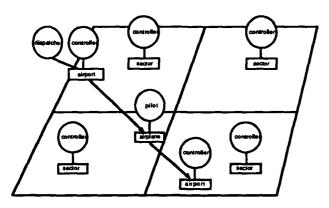


Figure 2: Relationship Between Simulation Objects

Discrete event simulations run within a common time base, with which they synchronize their actions. For example, this time base is used to synchronize pilot processes executing the Sleep_flight() procedure. Since the iPSC/2 system has no global clock, a global software clock is synthesized by Interwork II [8]. method used by Interwork II has the advantage of relative simplicity comparison to the Time Warp approach [4]. It also has the flexibility necessary to capture the simulation model's dynamic object creation, unlike the Chandy-Misra-Bryant [3] approach. However, in order to achieve substantial parallel speedup, this method requires the availability of many, distributed simulation events to process at each simulated time. The current implementation uses a floating simulation clock whose fine relatively granularity results in events to process at each simulated time. Future implementations will use integral clock periods with sufficient granularity to capture the simulation's behavior. should yield much greater approach parallelism for exploitation by the parallel system.

Maximal parallel speedup also depends on the distribution of the simulation events across the processing nodes at each simulated time. Future versions will explore the use of alternative (e.g.,

mappings of the interleaved) sector mappings to the nodes. Beyond this, dynamic load balancing algorithms may be needed to compensate for the dynamic redistribution of simulation events across the nodes as the simulation progresses. The use of a global object name space enables simulation objects to transparently moved between nodes load balance the simulation without changing the application code.

Summary

This paper has described a discrete event simulation model of air traffic flow within the United States. This simulation model is characterized by having a very large number of simulation entities and by its dynamic creation and destruction of these The simulation program was entities. written in an object oriented manner for Intel iPSC/2 distributed memory, parallel system using the Interwork II Concurrent Programming Toolkit. The iPSC/2 system has sufficient memory to hold the very large simulation program, and it has the processing power to exploit the simulation's parallelism.

The use of object oriented programming techniques led to a very straightforward simulation model. Interwork II's global object name space. indexed object synchronization paradigm, and global mechanism simplified in the model's implementation on the iPSC/2 system. particular, it provided the means automatically distribute the simulation objects among the processing nodes and to transparently access objects on remote nodes.

More work is needed to measure and improve the parallel speedup. In particular, the use of an integral simulation clock and dynamic load balancing techniques may prove useful in extracting more parallelism from the simulation.

References

- [1] Intel Scientific Computers 1988. iPSC 1/2 Technical Description, Beaverton, Oregon.
- [2] Hoare, C.A. R. 1974. Monitors: an Operating System Structuring Concept. Comm. ACM 17:10, (October): pp. 549-557.
- [3] Misra, J. 1986. Distributed Discrete-Event Simulation. Computing Surveys 18, no. 1 (March): pp. 39-65.
- [4] Jefferson, D. R. 1985. Virtual Time. ACM Trans. Prog. Lang. Syst. 7, no. 3 (July): pp. 404-425.
- [5] Block Island Technologies 1988. Interwork II™ Concurrent Programming Toolkit, Reference Manual, Portland, Oregon.
- [6] Bain, William L., 1988 A Global Object Name Space for the Intel Hypercube, Third Conference on Hypercube Concurrent Computers and Applications (Pasadena, Ca.., Jan 19-20).
- [7] Bain, William L. 1989. Indexed, Global Objects in Distributed Memory Parallel Architectures, Sigplan Notices, 24:4, (April): pp. 95-98.
- [8] Bain, William L. 1990. Parallel Discrete Event Simulation Using Synchronized Event Schedulers, Fifth Conference on Distributed Memory Concurrent Computers (Charleston, Apr. 9-12).

APPLICATION OF TRANSPUTERS TO AIRCRAFT SIMULATION AND CONTROL

D.J.Doorly S. Pesmajoglou

Aeronautics Dept. Imperial College London SW7 2AZ

Abstract

Techniques for implementing an aircraft simulation and control system on a network of transputers are described. Different parallelisation approaches are shown to be appropriate within each of the major constituent processes. The task of atmospheric turbulence simulation is used to illustrate the procedure.

Introduction

The Inmos Transputer is a processing element which allows great flexibility in the design of a distributed memory concurrent computer. The basic architecture of the T-800 series, comprising 32-bit CPU, on-chip FPU, 4K bytes of RAM, 4 bi-directional (20 Mbit/sec) links, and a 32-bit external memory interface, is now widely familiar, and well documented. The application of transputers to simulation in the aerospace industry, however, lags behind systems which are usually based either on bus-connected conventional microprocessors, or dedicated shared memory minisupercomputers. Previously the necessity of programming in Occam, or of writing a dedicated Occam harness to handle inter-processor communication, coupled with the lack of cross-development tools, proved a stumbling block which undoubtedly restricted the more widespread use of transputers. The development of parallel operating systems, and parallel versions of Fortran, C, Pascal and recently Ada for the transputer, should greatly assist in the translation of existing codes and provide a more open environment for code development.

This work is concerned with the conversion of a sequential Fortran code for aircraft simulation and control to run on a transputer network. The problem area provides a challenge for effective parallelisation, since the system functions as an aggregation of very disparate algorithms working in loose synchronisation.

The purpose of the paper is to examine simple parallelisation strategies for the various computational tasks which are performed by the separate functional units. The assessment of potential performance is intended to guide future work, which will concern synthesis of the complete parallel system. The paper is divided into four sections as follows.

The main components of the simulator are reviewed in the first section.

The second section discusses the use of information flow routes, and the amount of computation within procedures to guide the division of the system into a set of communicating tasks. The methods for software implementation and effective mapping onto the transputer network are outlined.

In the third section, the simulation of atmospheric turbulence is used as an example to illustrate how the parallelisation strategies may be applied. The problems in achieving effective parallelisation for a complex task, and of load balancing the parallel turbulence generator, simulator, and control system is then addressed.

Finally, in the fourth section, the procedures are reviewed, and the conclusions presented.

Overview

The simulator used for this work was designed to fulfill two roles. Firstly, it should allow the handling qualities of modified aircraft configurations to be studied, in both the linear (low angle of attack) and non-linear (high angle of attack) regimes. The second purpose is to assess the effectiveness of various control techniques to alleviate gust loads (due to atmospheric turbulence), and for the optimisation of advanced configurations in manoeuvring flight. The principal components of the system are outlined in Fig.1.

The simulator (1) models the response of the aircraft to inputs $\underline{\mathbf{u}}$ from the controller, and $\underline{\mathbf{d}}$ from the turbulence field. At each time step, the simulator outputs the updated complete state of the aircraft to a

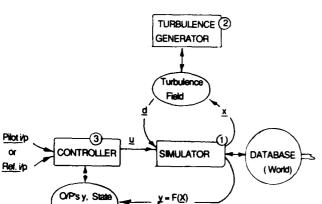


Fig. 1. Components of Simulator. Controller & Turbulence Generator have local Databases.

Estimates

database. This also holds details of the terrain, other aircraft, etc., and it may, in turn, be connected to a graphics display pipeline, via a ring control to which other simulators can be attached. (The graphics processing, and ring control are not of primary concern here; an effective parallel solution is documented in the Inmos applications notebook, [1], and their note 36 [2]. The simulator also outputs the position \underline{x} of the aircraft to a local database attached to the turbulence generator.

The turbulence generator (2) performs the twin functions of regularly updating the turbulence field, and of calculating the components of the local turbulent gust velocity $\underline{\mathbf{d}}$, when the aircraft is at a point specified by $\underline{\mathbf{x}}$. (The variation in turbulent velocity across the wing span introduces significant moments. It is necessary to include this, either by introducing averaged asymmetric turbulent velocities, (by integrating the turbulent velocities across the span), or for a more accurate treatment, to utilise a three dimensional turbulence field).

Finally, the controller (3), which also has a local database, accepts inputs from either the pilot, or a preset trajectory. It then compares this with the current estimate of the aircraft state (which it computes from the measured simulator output $\underline{\mathbf{y}}$), and sends a control $\underline{\mathbf{u}}$ to the simulator.

The first, obvious, stage in parallelism follows from the identification of the three major components. The use of specialised local databases, where possible, clearly distributes the communications load. Mapping the major tasks onto a network of distributed processors offers the major challenge, as considered

next.

Parallelisation Strategies for Transputers.

Most of the literature on parallelising sequential code is concerned with multi- processors of the shared memory type, and concentrates on DO LOOPS as the source of parallelism. Nearly 70 references to work in this area are listed in [3], which also reports some preliminary experiments on the feasibility of loop partitioning across a transputer chain.

At present, there is no efficient, autoparallelising conventional language compiler suitable for a distributed memory concurrent computer. As described in [4], to achieve this requires some information to be specified on the placement of data. The present parallel compilers for the transputer essentially mimic the underlying Occam, in concentrating on the distribution of tasks, to which the necessary inter-task message handling must be added. Although improvements in compilers (particularly C), and operating systems are being made, translation of sequential code is still a "hands-on" approach. Fortunately though, it is not necessary (particularly using the farm approach discussed below) to explicitly specify every realisation of a concurrent task. A knowledge of the relative performance of the transputer at processing, communications, and the speed ratio for on-versus off-chip RAM is however, obviously required. The transputer has the ability to communicate across links, whilst concurrently executing a second process, although each link transfer in all cases occupies the processor for a "setup" time. Decisions regarding the optimal parallelisation thus evolve by balancing the length of data packets sent, the setup times, as well as the connection topology, and the ability to utilise parallel execution threads, as discussed in [5].

A PC-hosted network of eight T-800 transputers using the 3L parallel Fortran compiler was used in this study, although future development work and implementation will use a Meiko Computing Surface, with an initial complement of 32 T-800 transputers.

The techniques for parallel processing may be grouped into three principal paradigms: the geometric array, the algorithmic pipe, and the processor farm. It is chiefly the latter two of these which have been applied to the part of the work reported here. (The geometric array is clearly advantageous for the display processing, and is also suitable for a turbulence generator more complex than that chosen as an illustration below.)

Algorithmic pipelines

In the algorithmic pipe, each task is first examined to establish the routes along which the data

processing flows, and the connections between routes. A single section of data flow, between input to the task, and output from the task is then treated as an algorithmic pipeline. There are, however, two major difficulties with the algorithmic pipeline which have been encountered.

Firstly, for any given task, there is a limit to the number of pipelined sections into which it may efficiently be divided. As the number of pipeline sections is increased, the work done in each section decreases, whereas the communication between tasks tends to increase, since more intermediate results need to be transferred. Consequently computation/communication ratio for each section diminishes rapidly. Although the transputer is capable of concurrent computation and communication in parallel threads once the communication link engines are started, a fixed time interval is required to set up the link engines. The subdivision of each task is relatively coarse grained as a result, although since there is a large number of tasks, this does not prove to be restrictive in practice.

A second problem is that one or more sections in the pipeline may require far more computation time than other stages, leading to execution bottlenecks, and consequent low parallel efficiencies. Sub-division into more sections is often not possible, for the reasons above. In this case, possible solutions are:

- (i) use a faster processor (a valid option now that heterogeneous processor networks (for example T-800 and i860) are available.)
- (ii) Effectively "widen the pipeline" at the bottleneck, using an array of transputers.

Two examples taken from the simulator illustrate the application of algorithmic pipelines to the problem area.

Fig.2a

Simple Algorithmic Pipeline.

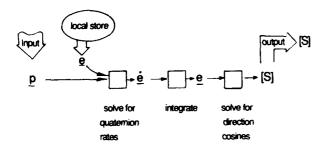


Fig. 2a shows a simple process, that of calculating a new value of the direction cosine matrix [S] from the rotation rate vector. The computation follows three stages, and is thus divided into three communicating processes, marked by boxes. Since the computational requirements of each process is roughly comparable, each task may be placed on a separate processor. The simplest approach to resolving computational imbalance, which can sometimes be applied, is to merge those requiring least, and sub-divide into a cascade those requiring most computation, (eg. for a multistage integration scheme).

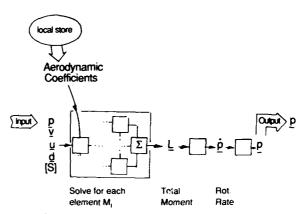


Fig. 2b.
Use of Embedded Geometric Array in Algorithmic Pipe.

To update the rotation rate vector is slightly more complex, (fig. 2b), particularly in a more exact simulation where the variation in turbulent gust velocity across the wing span is included in the model. The simplest approach considered effectively divides the wing span into parallel strips, and evaluates the contribution of each strip, to the total, together with an overall correction. Since this computation takes by far the longest time in this section of the simulator, branching the pipeline, or using an array to compute the elemental contributions may be used. This provides an effective solution, and is suitable for direct mapping as a transputer array embedded in a linear pipe.

Processor Farms

The processor farm uses a different approach, and may be of the data farming or task farming type. In data farming, identical copies of a program ("workers") are distributed across a set of processors, which are under the control of a master program ("farmer"). The farmer sends data packets to the workers, and collects their results. (Task farming requires the master to

dynamically load tasks onto processors via the operating system, and is not considered here). A "router task" is placed on each transputer node of the network, to pass the work packets to the available processors. It is not necessary to specify the number of processors or configuration, as this is determined at load time.

A problem with current versions of the transputer is that messages routed through an intermediate transputer from one node to another results in delays. Consequently, as the routing of messages is generally less efficient and the communications overhead larger than in the case of a fixed geometric partitioning (where the latter is an alternative), one often achieves disappointingly poor parallel efficiencies. implement a general farm-based approach apparently still necessitates programming in Occam, however for this work the 3L Fortran library of farm handling procedures was used. The results obtained in a number of different applications indicated that, for this implementation at least, high levels of parallel efficiencies could only be attained if the worker tasks are computationally intensive.

Turbulence Simulation and Aircraft Control.

The generation of a single component of a one dimensional (ie. no variation in y or z) turbulent gust velocity field is considered first. Let d(x) denote the vertical component of the disturbance velocity along the x-axis, (the direction of flight). It is required that both the spatial frequency content, and the probability distribution function of d(x) should closely match that measured in real turbulence. The method used 3) requires three independent uniform random number sequences. (The procedure used for the random sequence generation is the standard linear congruential method, with an additional randomising shuffle table, [6]). Firstly, the random numbers are processed to obtain a Gaussian distribution. An approximately Gaussian distributed random sequence is most simply obtained by forming the subsequence consisting of the mean of n terms of the original uniformly distributed sequence, where n is at least 12.

The processed sequences serve as independent white noise sources n_1, n_2 , and n_3 , and each source then passes through a linear filter. The output from any of these filters has the correct spectral distribution of frequencies, but the spatial distribution is too homogeneous, as shown. To correctly represent the non-Gaussian "patchy" nature of real turbulence, the outputs of the linear filters are combined non-linearly, [7].

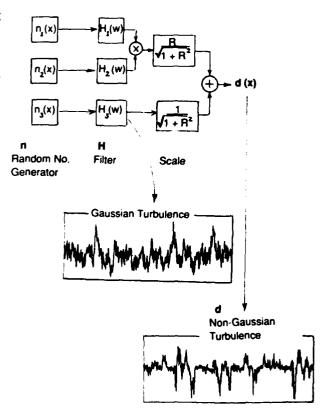


Fig. 3
Generation of Simulated Turbulence

The longitudinal and lateral turbulent velocity components are generated in a similar fashion, with slightly different linear filters. To account for the effect of lateral variations in the turbulent velocity field, the simplest approach is to generate an additional two velocity components. The procedure is exactly as before, (though with appropriate filter characteristics). These velocity components account for the integrated, moment inducing effect, of instantaneous spanwise asymmetry in the longitudinal and vertical turbulent gust velocities, [8]. Thus fifteen random numbers generators may be required to provide the five turbulent velocity components required for a two dimensional (strictly quasi-two dimensional) turbulence field simulation.

The obvious technique which may be used to apply the linear filters is the FFT. The spectral distribution of real turbulence however may be approximated sufficiently accurately for simulation by a simple rational filter, such as:

$$H(w) = A/(1+B.jw)$$

with A,B constants, and w the frequency variable. The Z-transform may then be applied to derive a recursive filter, such that the i-th turbulence velocity d_i is given in terms of previously filtered values d_{i-1}, d_{i-2} , and previous unfiltered inputs w_{i-1}, w_{i-2} , by a relation such as:

$$d_i = k1.d_{i-1} + k2.d_{i-2} + k3. w_i + k4. w_{i-1} + k5.w_{i-2}$$

where k1, k2 etc. are constants

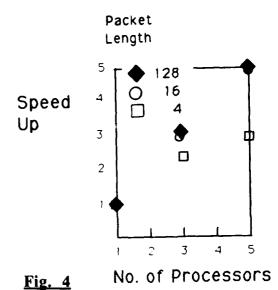
Clearly this provides a considerable computational saving. Furthermore, by relating the position x, to the time t, the turbulent velocity components may be generated as a time series, (ie. at each time step in the solution of the equations). This is the approach normally used for the implementation of one dimensional and quasi two dimensional turbulence generators. For an efficient transputer implementation it is not appropriate to attempt to generate each value just when it is required for the reasons considered next.

The generation of random numbers is well suited to the data farm aproach, as it is only necessary to start each generator with different parameters and initial seeds. The generator is formulated as a worker task, under the supervision of a master task, and a copy of the worker resides on each transputer including the root node. The master task, which resides on the root transputer, distributes the initial data to the generators, and collects the results. The sequence of random values produced by the generators may be routed back to the master task in packets comprising either a single value, or up to 256 values, (in the current 3L release).

It is not efficient, however, for each worker to send back one random value at a time, since the wasted processor time required to initiate a communication across a hardware link then becomes very significant in relation to the time spent in computation. This is illustrated in fig.4 in which the performance of a farm based Gaussian random number generator is compared for varying result packet lengths. Consequently each worker generates a long array of values which are transmitted back to the master as a single packet.

The workers continue asynchronously generating and transmitting packets without further intervention. Simple double buffering may be used in the output from the master task, to emulate continuous generation, if values are required on a pointwise basis.

In the implementation of the turbulence generator, it is possible to lump the filtering together with the random number generation on the worker tasks. This has the potential of ensuring that the worker tasks are highly computationally intensive.



Performance of Farm Algorithm: Effect of Length (in integers) of Results Packet.

Since the filters differ for each velocity component however, a farm-based implementation of this form would require non-identical workers. Although this could be accomplished, it adds complexity, and is not easily realisable for a general transputer network. In any event, the computational overhead required for a simple recursive filter is very low, and may be placed entirely on the master task, without noticeably reducing the high efficiencies achieved.

To implement a full three-dimensional turbulence field generator, requires the use of the FFT for the spatial filtering. Although it would be very useful to combine a processor farm for the random noise generation, with a geometric array to accomplish the spatial filtering in a hybrid parallel system, this option is not effectively supported for PC-hosted systems in the current 3L release. For a complete simulation therefore, it is necessary to specify the entire network, effectively as a geometric configuration. Load balancing between the resources required by the random generators, and the filters is thus fixed, as a result of a process of trial and error. It is difficult to gain a satisfactory (or even sometimes any) gain in performance each time a single transputer is added to a completely geometrically partitioned network. The simplest procedure is clearly to add the transputers to increase the performance of one part of the network (eg. the filter stage) at a time. Future work using the Meiko should allow the implementation of hybrid strategies.

The results show, however, that it is possible to obtain high parallel efficiencies in the generation of one

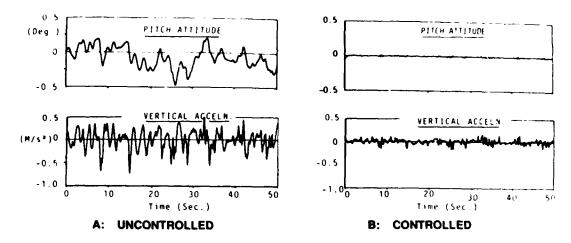


Fig. 5 Simulated aircraft response to atmospheric turbulence

Discussion and Conclusions

and quasi-two dimensional turbulence.

For aircraft control, simple PID controllers, predictive, and linear optimal controllers are being considered. Fig. 5 shows the application of an optimal controlled (applied to stabilise the longitudinal dynamics only, ie. velocity components u,w, pitch angle q, and rate q, and augmented by the integral of pitch). For the linearised state variable equations,

$$x = [A]x + [B_u]u + {}^{t}B_d]d$$

The control vector \mathbf{u} is obtained from

$$\underline{\mathbf{u}} = -\{\mathbf{R}\}^{-1} [\mathbf{B}_{\mathbf{u}}]^{\mathrm{T}} [\mathbf{P}] \underline{\mathbf{x}}$$

where P is the solution of the steady-state Ricatti equation,

$$[P][A] + [A] ^T [P] - [P] [B_u] [R]^{-1} [B_u] ^T [P] + [Q] = 0$$

and [Q] and [R] are the weighting matrices associated with the state-space vector and control input respectively.

The solution of the above matrix equation is accomplished by converting to an eigenvalue problem, forming a matrix of eigenvectors, and performing an inversion.

The parallelisation procedures which have been considered to date to obtain the control vector **u** comprise a coarse algorithmic parallelisation of the major stages of the solution algorithm. As the techniques are as described above this will not be discussed further. The potential efficiencies attainable are high, for a coarse discretisation is. Currently approaches for a fine discretisation are being considered.

Both algorithmic pipeline and processor farm approaches have been applied to the parallel decomposition of an aircast flight simulation and control system. Currently the techniques employed for load balancing are very much of a "hands-on" nature, for the geometric and algorithmic parallelisation strategies. To allow more efficient general parallelisation, the problem of through communication delays needs to be resolved on the hardware side, whilst the ability to mix paradigms would be very useful. It is anticipated that the next generation of transputers will solve the routing problem, and also facilitate effective general parallelisation, rather than adherence to a rigid paradigm. Hopefully this should allow the obvious benefits of the transputer to be more generally employed for this application area.

References

- 1. Inmos Transputer Applications Handbok, 1989
- 2. Inmos technical note no. 36
- van Santen, P. & S.K. Robinson. "Parallel Extraction Techniques and Loosely

Coupled Target Systems" Software For Parallel Computers 1989

- Lake, T "Distributing Computations" Software For Parallel Computers 1989
- Pritchard, D. "Performance analysis and measurement on Transputer Arrays" Software For Parallel Computers 1989.
- Knuth, Donald E. "Seminumerical Algorithms, 2nd ed. vol. 2 of "The Art of Computer Programming" pp. 116, 1981.
- 7. Reeves, P.M. et al.

NASA CR-2451, 1974

 Gerlach, O.H. & Baarspul, M.
 Rept. VTH-139. Delft University of Technology, 1968

Simulation Of An Urban Mobile Radio Channel on the Myrias SPS-2

M. Fattouche ATRC Affiliate Professor University of Calgary Calgary, Alberta Canada L. Petherick Myrias Research Corporation Edmonton, Alberta Canada

A. Fapojuwo ATRC Postdoctorate Fellow University of Calgary Calgary, Alberta Canada

Abstract

Combining techniques in an urban mobile communications channel are simulated using a modified Hashemi model. The parallel implementation of this simulation on a distributed memory, MIMD parallel architecture is discussed. Combining techniques suitable for the first generation digital cellular communications system in North America are analyzed. Scalability and overall performance results on the Myrias SPS-2 are presented.

Introduction

In mobile radio, energy can travel from the transmitter to the receiver via more than one path. This "multipath" situation arises because of reflection and scattering from buildings, trees and other obstacles along the path. At the receiver, the radio waves combine vectorially to give a resultant signal which can be small. When this occurs, the signal is said to be subject to fading. Moreover, whenever relative motion exists, there is a Doppler shift in the received signal. The combined effect of fading and Doppler produces a received signal with an amplitude and phase that changes quite substantially with time.

This poster session describes the simulation of the first generation digital cellular communications system in North America [1], together with a suitable technique for carrying out the simulation on the Myrias SPS-2, a parallel processing architecture. Bit error rate curves resulting from the simulation are presented. Scalability results and overall performance of the simulation on the SPS-2 are presented. Conclusions derived from results obtained using this simulation, as well as the suitability of running this simulation on the SPS-2 are discussed.

The Digital Radio Simulator

The combined effect of Doppler and fading is simulated using a modified Hashemi model [2], [3]. A block diagram of the channel simulator is given in Figure 1. The transmitter consists of a precoder and a transmitting filter corresponding to the first generation digital cellular communications system in North America [1]. The precoder maps the information sequence $\{\alpha\}$ into $\pi/4$ -shifted Quadrature Phase Shift Keying (QPSK) and the transmitting filter corresponds to a square root spectral raised cosine filter with a roll-off factor of 0.25.

The channel in Figure 1 corresponds to one that would be used in a sparse high rise urban environment at 800 MHz carrier frequency, with the vehicle traveling at 50 km/hr. The delay spread is limited to $7 \mu \text{sec}$, which corresponds to intersymbol interference over only one adjacent symbol. This yields a symbol rate of 24 ksymbols/sec.

The receiver can accommodate a receiving filter identical to the transmitting filter, an adaptive equalizer, and/or a Viterbi algorithm (VA). The equalizer can be either a decision feedback equalization (DFE) or a simple linear equalization (LE). The adaptation algorithm associated with it can be either a least mean square (LMS) or a recursive least square (RLS) algorithm.

Parallel Implementation

Digital radio channel simulations are done on the Myrias SPS-2. The SPS-2 utilizes a distributed memory, MIMD parallel processing architecture. A parallel implementation of the simulation is done in the following manner. The first level of parallelism implemented is to analyze different system parameters/operating conditions in parallel. For each set of these operating conditions, the simulation is done over several city blocks. Calculations within a given city block are independent of all other locations; hence are done in parallel. Several signal to noise ratios (SNR) are simulated within the channel in each city block. Calculations for each of the SNR are also done in parallel. Thus, 3 levels of parallelism are identified, and implemented.

Bit error rates predicted using the simulation are discussed in the next section.

Simulation Results

Using the operating conditions associated with Figure 1, simulation results were obtained. Figure 2 displays 5 bit error rate curves versus E_b/N_0 , where E_b is the average bit energy and $N_0/2$ is the magnitude of the two-sided power spectral density of the additive white Gaussian noise. Curves A and B correspond to a receiver with DFE, curves D and E to a receiver with LE and curve C with no equalization. Curves A and D correspond to a receiver with no receiving filter, and curves B, C, and E to a receiver with a receiving filter. Run times for the simulation and scalability results, using the Myrias SPS-2 are presented in the next section.

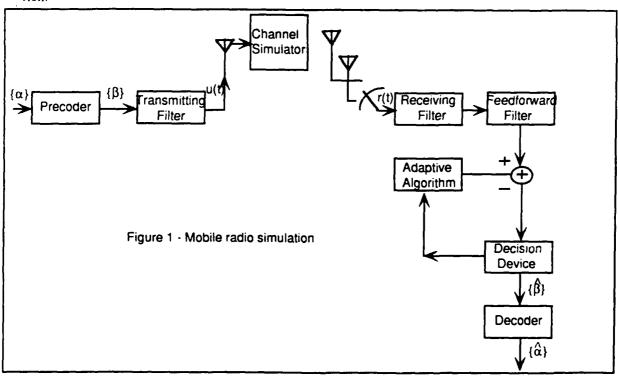
Performance Results

The channel is simulated using 9 to 378 processors on the SPS-2. Absolute performance results are measured using 9 to 108 processors. Scalability results are measured using 54 to 378 processors. It should be noted that no changes to the code or the executable are required to run the program using different numbers of processors.

Figure 3 shows absolute performance results for two sets of operating conditions. The first set of operating conditions includes a DFE with no receiving filter. The second set contains both a DFE and a receiving filter. Each channel is simulated over 6 city blocks. 9 SNRs are evaluated. Using 9 processors, a run time of 13 hours and 27 minutes was measured. The same simulations were done using 18, 54, and 108 processors.

The run time decreases by a factor of 11.5 when the number of processors is increased by a factor of 12, from 9 to 108.

Scalability results are presented in Figure 4, using 54 to 378 processors on the SPS-2. Scalability is defined here to be constant work (computation) per processor. One set of operating conditions is simulated using 54 processors, and 2 sets on 108 processors.



Results are also presented for simulations performed using 216, 324, and 378 processors. Run times (elapsed time), non-dimensionalized using a run time of 1 hour and 9 minutes on 54 processors, vary from 1.01 on 108 processors to 1.10 on 378 processors. Using 54 processors, 96% of the total run time is actual user (CPU) time. 1% is used by the operating system. Idle processors account for the remaining 3% of the elapsed time.

Some processors are idle during the following parts of the simulation. Initially, only one processor is used when the system parameters for each datafile are being set up. Processors are also idle while the simulation over each city block is being established.

Using 378 processors, idle time accounts for 12% of the run (elapsed) time. User time (CPU) is 87%. The remaining 1% is used by the operating system.

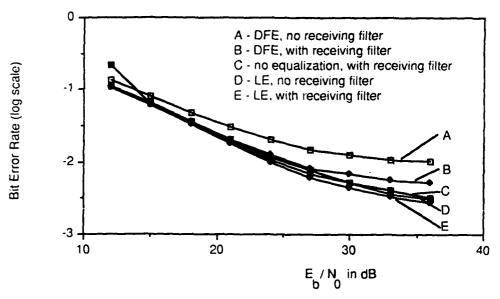


Figure 2. The bit error rate for urban land mobile channels

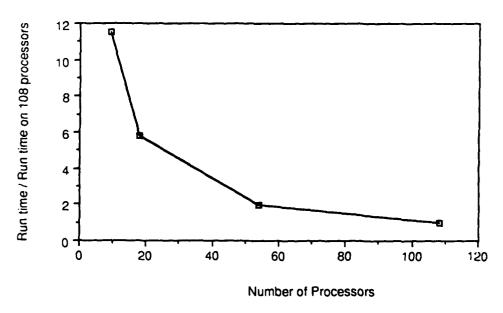


Figure 3. Scalability - fixed problem size

Conclusions

From Figure 2, it is seen that DFE offers a poorer performance than LE, and should therefore be avoided. It is also seen that LE offers a slight improvement in performance over having only a receiving filter. However, the improvement is very small at bit error rates of $10^{(-2)}$. For that reason, it is believed that equalization is not needed.

The VA is not displayed in Figure 1, since it was found that with a delay spread limited to $7~\mu \rm{sec}$, the channel is essentially flat. In other words, there is no need for a VA for such a channel. Moreover, it is found that with a vehicle travelling at 50~km/hr, the channel is fast fading, and the LMS algorithm fails to track the rapid amplitude and phase variations of the channel.

The Myrias SPS-2 is a very suitable computer platform for doing this type of simulation. An existing code, written in Fortran 77, was

ported to the SPS-2 with very little effort.

Testing and evaluation of various techniques used in the channel simulation program are done rapidly on the SPS-2. Use of the simulation on the SPS-2 enables rapid evaluation of many configurations before the final real world experiments are conducted.

References

- [1] Cellular System, January 1990, Dual-Mode mobile station-base station compatibility standard, EIA/TIA, Project Number 2215, Electronic Industries Association.
- [2] Hashemi, H., 1979, Simulation of the urban radio propagation channel. IEEE Trans. Veh. Tech., vol. VT-28, pp. 213-225.
- [3] Turin, G.L., 1972, A statistical model of urban multipath propagation, IEEE Trans. Veh. Tech., vol. VT-21, pp. 1-9.

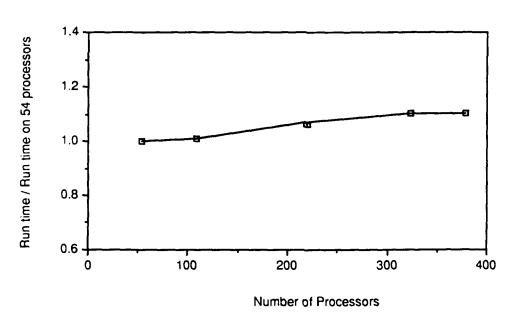


Figure 4. Scalability - constant work per processor

PORTABLE ASTEROIDS ON HYPERCUBE OR TRANSPUTERS

Alex W. Ho and Geoffrey C. Fox

Concurrent Computation Program 206-49, California Institute of Technology, Pasadena, CA 91125, USA

Abstract

A multi-player 3D Asteroids video game designed to be used as a testbed for evaluating controller algorithms was described in [1.] The original version of the game and a separate interactive 3D graphics interface for a human player were implemented, based on CrOS III and VERTEX, on an NCUBE-1 hypercube equipped with a parallel Real-Time Graphics board. The Asteroids and interactive graphics interface programs are examples of parallel programs which communicate with each other in a space-shared multi-processor environment.

We have successfully ported the Asteroids and the interactive graphics interface to run on NCUBE using ParaSoft EXPRESS. The new version of these programs were further ported to run on a SUN 386i with an add-on Transputer board. We present general design considerations that enable easy migration of communicating parallel programs to any other hardware platform that runs EXPRESS. We also report specific experience of porting Asteroids and an associated interactive player interface program on an NCUBE hypercube to a SUN 386i Transputer-based system, with no modification of codes.

Introduction

Code portability is a major concern for people who writes programs, and especially so for those who implement computation intensive algorithms. Scientists would like to run their specialized codes, without modification, on faster computers whenever they are available. Ample examples can be found in the fields of computational fluid dynamics, chemical dynamics, and in quantum chromodynamics, just to name a few.

There is another class of computation intensive programs which compete or cooperate with one another within a simulated organizational structure. Usually, these are programs which implement artificial intelligence, decision-making algorithms. An example of a simulated organizational framework is a game environment with a game manager program which coordinates the actions and competitions of multiple player programs via message-passing. The "players" and the game manager can benefit from parallelization. However, it is difficult to develop portable codes for these communicating parallel programs which does not only require interprocessor communication within each program but also communication among different programs.

There is a proliferation of small parallel computer systems for tutorial and experimental purposes. Among these, the Transputer-based system is a popular one. We present general system design guidelines which enable easy porting of the game environment to other hardware platforms that are supported by EXPRESS, and discuss specific experience in the porting of the NCUBE Asteroids to SUN 386i Transputer-based system.

Why A Game?

There have been mammoth interest in research on intelligent controller algorithms which can perform tasks that normally require human supervision for decision making. Some examples of such tasks are navigation control and multi-target tracking [2, 3.] Intelligent algorithms are in general very computation intensive. Moreover, there are no effective ways to evaluate or compare the performance of these algorithms, either running alone or simultaneously.

A dynamic game which contains the features of randomness, secrecy, incomplete and noisy infor-

mation, as well as limited resources of the players would provide a natural arena for these algorithms. Such a game generates a consistent, dynamically evolving environment for the participating player programs which are implementations of various algorithms for some simple, well-defined objectives. It is also essential that such a game be implemented on a powerful computing environment so that computation intensive algorithms can compete fairly in real-time.

Asteroids

The Asteroids arcade game is a single player game which features a spacecraft traversing a 2D toroidal space with inert, moving celestial bodies of various sizes. Given an interactive graphics display and button-controlled interface, a human player can maneuver a spacecraft to turn, thrust, vank, or to fire missiles. The objective of the game is very simple. It is to destroy as many asteroids as possible without being hit by them. Large asteroids split into multiple smaller ones when hit by other asteroids, missiles or spacecraft. A spacecraft is destroyed when hit by any objects. Since the Asteroids game is conceptually simple, we have chosen to implement it, with some enhancement, as a testbed for the evaluation of intelligent algorithms which are developed specifically to achieve the game objectives.

We have implemented a 3D Asteroids game environment on an NCUBE hypercube which was equipped with a parallel graphics board [1.] The software system was based on CrOS III and VERTEX. The implementation of Asteroids on a space-shared concurrent processor makes it easy to compare performance of different algorithms that are assigned to a common task at the same time. Preferably, an intelligent algorithm in use is parallelized to take advantage of the multi-processor architecture for efficiency. Otherwise, it will be trivial to modify a sequential program so that it will run on one node of a concurrent processor, and still be able to take part in the game.

The enhanced Asteroids game models spacecrafts and asteroids, governed by physical laws, traversing a 3D toroidal space. Unlike the arcade game, spacecrafts are not destroyed immediately when collide with other flying objects. They only lose 'energy' which is used as an index of cost. If a player's spacecraft is out of energy, that player is out of the game. 3D Asteroids is designed to accommodate multiple 'players'. All players do not have to join the game at the same time. At any time when the game is running, the game program is capable of adding new or removing existing 'players'. This arrangement allows a real-time competition among the different 'players' who are subjected to the same global conditions and games rules, but are occupying different locations in the 3D toroidal space.

Overall Design of Asteroids

One way to look at the Asteroids game system is to treat the game objectives as the objective functions of an optimization problem which is constrained by the imposed game rules. The user-supplied algorithms, including the interactive player's intelligence, implement different approaches to solve the posed problem. Therefore, it is essential that the game can support multiple players for the purpose of direct comparison of several algorithms. This also makes the game more realistic and exciting.

All programs that are involved in Asteroids do not make any assumptions about the underlying hardware environment, and are classified by functionality into three categories. They are the 'game driver', 'player', and 'graphics driver' programs.

The 'game driver' is the core of the game. Only one copy of the game driver is needed at all time. The primary entities in the game driver are objects like spacecraft, missiles, and asteroids. It implements rules of the game, processes player requests, and evolves game objects in time.

There are two types of 'player' programs. An interactive player program implements a 3D graphics interface for a human player to control a spacecraft, while a batch player program implements an intelligent algorithm to take over the responsibilities of what a human player is supposed to do during the game. A player program is isolated from the rest of the game so that any modifications of it will affect the performance of an individual player only, and has no effect on the operation of the game itself.

A 'graphics driver' is an interface between player programs and the graphics hardware. It provides the low-level graphics operations for player programs and isolates them from the ever-changing graphics hardware. An interactive player program certainly needs graphics support because a human player relies on the visual-oriented display to make decisions. A batch player program has the option of using graphics display for the convenience of the observers of the game.

Hardware Considerations

The first version of Asteroids was developed for an NCUBE hypercube with a Real-Time Parallel Graphics Board which has 16 NCUBE processors, and uses Hitachi HD63484 Advanced CRT Controllers (ACRTC.) The processors on the graphics board will be called graphics nodes, and those on an NCUBE hypercube will be called array nodes for nomenclature convenience.

Figure 1 is a block diagram of an NCUBE with a parallel graphics system. The control processor of the entire system is an Intel 80286. Two distinct features of the graphics system are that the 16 graphics nodes are capable of communicating with each other, or with the array nodes using high-speed I/O channels; also, signals from the graphics tablet can by-pass the control processor and reach the graphics board directly via a RS-232 port at 19200 band.

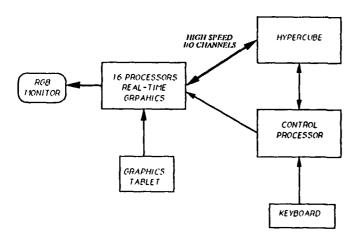


Figure 1: A Block Diagram for an NCUBE-1 with Real-Time Graphics

A graphics node can issue a graphics command, by sending a message to the 80186 on the graphics board, to initiate a DMA transfer of pixel data in the local memory of the graphics node to the frame buffer of the display monitor. Local memory of the graphics nodes are mapped to the frame buffer in alternating 2-pixel wide strips. A DMA

transfer takes 1/30 second. However, altering the data in the frame buffer while a DMA is in progress usually produces an unpleasant among of flicker.

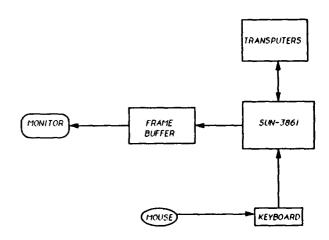


Figure 2: A Block Diagram for a SUN 386i with an add-on Transputer Board

Of the 128K local memory available in the graphics nodes, about 20K is used by GRAPHOS (a nucleus similar to VERTEX.) A single buffer for each graphics node is 48K. If 2 consecutive 1/16 frames of display are to be computed by each of the 16 graphics nodes before calling a DMA transfer, the executable graphics processing program on the graphics nodes has to be smaller than about 8K. It is unreasonable to expect any realistic graphics programs to occupy only 8K memory space. Therefore, it is very difficult to make use of the 2-Mbyte frame buffer for real-time double buffering.

A block diagram for a SUN 386i is shown in Fig. 2. It is a much simpler configuration because it does not have a parallel graphics system. The SUN 386i acts as the control processor for an addon multi-processor Transputer board. All input devices are connected to the SUN 386i. There is no direct I/O channel from the Transputer board to the frame buffer which talks to the 386i only. This hardware configuration will not support real-time animation. However, the speed of graphics display can always be improved by adding specialized graphics hardware later.

System Software Considerations

There is no established standard for the new parallel computer languages, programming methodologies and operating systems. We have chosen to implement the new version of Asteroids on an NCUBE and a SUN 386i Transputer system using ParaSoft EXPRESS. The few reasons behind this choice are that EXPRESS is portable, simple, efficient, and CrOS compatible. Any carefully written EXPRESS applications can be migrated separately from one hardware platform to another relatively easily, as long as the computer system runs EXPRESS.

Implementation Guidelines

Portable and efficient intra-program communication is easy because they are provided by EX-PRESS functions which are already available for a wide range of parallel computer. However, porting a set of parallel programs which space-shared a concurrent processor and communicate with one another is not as straight-forward. Some operating systems, like VERTEX on NCUBE, do not allow a parallel program to send messages outside its own allocated sub-cube to another sub-cube within the main array. Also, there are hardware dependent codes such as those for graphics display. Efficient graphics are hardly portable because it involves too much hardware specific programming.

In order to make Asteroids portable, i.e., to run the same game driver and its associated player programs unchanged on different hardware platforms, an extra layer of software which contains two modules is introduced. These tv modules, INTERCOM and POLYCOM are small user-level libraries which provide player programs with the capability to communicate with the game and graphics drivers, respectively. We have implemented INTERCOM and POLYCOM on NCUBE-1 with a Real-Time Graphics board, and SUN 386i with a Transputer board.

Implementation of INTERCOM

The migration of the CrOS-based Asteroid to EXPRESS-based is straight-forward and does not deserve further discussion. We start the discussion with the implementation of INTERCOM.

Common to most distributed-memory concurrent computers is a control processor (CP) which usually runs a version of Unix or Unix-like operating system such as SUN-OS on a SUN 386i, and AXIS on an NCUBE. These operating systems support multi-tasking on the CP. A simple approach

to implement INTERCOM is to make use of Unixstyle pipe. Even though AXIS does not provide system support for pipe communication on the CP, it is not difficult to implement such a mechanism. Using pipes, the game driver and the player programs which run on the same space-shared parallel computer can communicate with one another on the CP. However, this method is very inefficient and is not suitable for real-time simulations, especially when the CP has to perform many other tasks besides handling the game processes. It is more acceptable if inter-program communication takes place within the parallel computer or via special high-speed I/O channels.

On an NCUBE-1 with a Real-Time Graphics board, inter-program communication can take place via the graphics board which has 16 high-speed I/O channels to the main array. Since VERTEX only checks on the destination of messages that originate from a processor in the main array, we made use of the graphics board to handle message routing to different sub-cubes of the NCUBE hypercube. When a player program (in a sub-cube) sends a message to the game driver (in another sub-cube,) the message is actually being routed through the graphics I/O board. The high-level INTERCOM library provides the service transparently with the help of a set of message forwarding routines in the FWDLIB library which has to be downloaded to the 16 graphics nodes before loading the game driver and the player programs onto distinct sub-cubes in the main array.

Since EXPRESS does not check on the destination of a message and it is the native operating system running on each Transputer processor, inter-program communication between player programs and the game driver can take place entirely within the Transputer network. For portability, an equivalent INTERCOM library is written on top of EXPRESS for a SUN 386i Transputer-based system. In this case, no FWDLIB library is needed.

The INTERCOM library for the game is very simple. There are only four routines available. At the beginning of a player program, a call to play_init() will register the player with the game driver. The game driver will be able to find out the number of processors a player occupies, and assign player number. When a player makes a call to read_state(), a new update of the environment will be returned. All nodes of a player will receive the same message from the game driver. If a player

wants to send a move to the game driver, it makes a call to send_moves(). For a player program which expects to receive input from the keyboard, a call to get_keys() will fill a designated buffer with all the keystrokes received so far, and the number of keystrokes placed in the buffer will be returned.

The FWDLIB library is implemented for the NCUBE-1 with parallel graphics only. It provides communications between arbitrary nodes in the main array, regardless of whether they are in the same allocation group or not. The library maintains 16 communication channels, each of which stores the addresses of two sub-cubes in the main array as well as the addresses of a particular node in each sub-cube as a receiver. If an array node in one of the two sub-cubes sends a message with a call to fwd_msg(), it will be sent to the receiver in the other sub-cube, where it can be read with a call to get_msg(). An array node can identify itself as a receiver and its allocation group as the sub-cube by calling attach_to_channel(). To detach both communicating sub-cubes from a specific communication channel, the parallel programs running in the two sub-cubes have to call clear_channel().

The INTERCOM library on NCUBE makes use of FWDLIB library implicitly. A player program using INTERCOM can communicate with the game driver without using or the need to know any of the FWDLIB routines.

Implementation of POLYCOM

There is a significant difference between the NCUBE-1 and SUN 386i graphics hardware, as can be seen in Fig. 1 and 2. A user of the Asteroids system who's main concern is to develop intelligent algorithms to play the game would not want to spend too much time in experimenting different graphics display strategies, not to say to deal directly with the graphics hardware at a very low-level. We have developed a parallel polygon graphics drivers for the NCUBE Real-Time Graphics board and an equivalent Sunview-based graphics driver for the SUN 386i.

On an NCUBE, player programs run on distinct sub-cubes in the main array, while the parallel polygon graphics driver runs on the 16 graphics nodes on the Real-Time Graphics board. On a SUN 386i system with no specialized graphics hardware, player programs run on the Transputer nodes, and the graphics driver runs on the CP, i.e., the

SUN 386i. Since a graphics driver and player programs run on different processors with no shared-memory, the player programs have to send drawing commands via messages.

While the graphics drivers hide all hardware details and provide 3D polygon drawing capabilities for the players, POLYCOM is a small library which furnishes a consistent set of user-level routines for player programs to communicate with the graphics driver. Player programs using POLYCOM can send drawing instructions to the graphics driver without knowing where it is.

POLYCOM supports drawing points, lines, and filled polygons. Simple functions like point(), pointset(), line(), and polyline() are available. It can draw background stars by star() or starset() for any space games. It also supports polygon drawing by the the function calls poly() or polyset() which draws a collection of one or more filled or wire-frame polygons. Fundamental graphics routines like ginit() for initializing the graphics library, reseting the clipping boundaries, and clearing the screen, setclip() for setting the clipping boundaries, setcolor() for changing the RGB values of a palette entry, dma() for making drawing visible by sending images to the frame buffer, gcmd() and gcmd_nodma() for executing the accumulated drawing commands with or without automatic calling of dma() are also provided by POLY-COM.

Asteroids on NCUBE and Transputers

The Asteroids game was implemented both on NCUBE with parallel graphics and SUN 386i with a Transputer board. It uses EXPRESS, INTERCOM, and POLYCOM for intra and inter-program communication. The overall relationships of the three category of programs and the communication among them are illustrated in Fig. 3 and 4.

Oval shape is used for a process, and rectangular box is used to differential the three types of programs in Fig. 3 and 4. The top level of a box indicates the type of program, while the lower levels show the libraries in use by the program. EXPRESS is not included in the boxes because we have assumed that it is available and is being used for programs that require intra-program communication. Bi-directional arrows in the figures indicate links for inter-program communication, while unidirectional arrows show the parent and child relationship of processes. Although there is a batch

player program in both Fig. 3 and 4, it has not been developed yet. The two figures just assume that competing player programs exist.

NCUBE

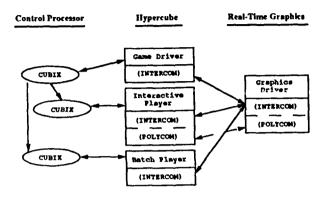


Figure 3: Schematic of program relationship in NCUBE Asteroids.

SUN 3861/Transputer

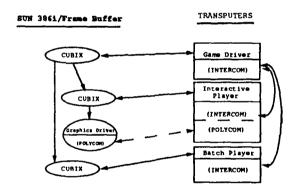


Figure 4: Schematic of program relationship in SUN 386i Asteroids.

Conclusions

We have presented guidelines to port communicating programs, both parallel or sequential, which space-shared a distributed memory concurrent processor environment. Specifically, we discussed porting a version of NCUBE Asteroids and an associated interactive graphics interface for a human player to a SUN 386i with a multi-processor Transputer board. We showed that, following the

general guidelines, both the Asteroids game driver and the associated interactive player programs can be migrated from an NCUBE-1 with a Real-Time Graphics board to a SUN 386i Transputer-based system with absolutely no change of codes.

Acknowledgement

This study is based on research work supported by the Joint Tactical Fusion Program Manager.

References

- [1] IIo, Alex W. and Fox, Geoffrey C. and Snyder, Scott and Chu, Diana and Mlynar, Ted, "3-D Asteroids Using Parallel Graphics on NCUBE: A Testbed for Evaluating Controller Algorithms," in Proceedings of The Fourth Conference on Hypercubes, Concurrent Computers, and Applications, pp. 1177-80, Monterey, Ca.; also in Caltech report C^3P -681b, 1989.
- [2] Ho, Alex W. and Fox, Geoffrey C., "Learning to Plan Near-Optimal, Collision-Free Paths," to appear in Proceedings of The Fifth Conference on Distributed Memory Computing Conference, Charleston, South Carolina; also Caltech report C^3P -881, 1990.
- [3] Gurewitz, Eitan and Fox, Geoffrey C. and Wong, Yiu-Fai, "Parallel Algorithm for One and Two-Vehicle Navigation," submitted to The Fifth Distributed Memory Computing Conference, Charleston, South Carolina; also Caltech report C³P-876, 1990.

A General Framework for Complex Time-Driven Simulations on Hypercubes

David L. Meier, Kathleen L. Cloud, Joan C. Horvath, Lynn D. Allan, Wayne H. Hammond Jet Propulsion Laboratory

Heath A. Maxfield California Institute of Technology

ABSTRACT

We describe a general framework for building and running complex time-driven simulations with several levels of concurrency. The framework has been implemented on the Caltech/JPL Mark IIIfp hypercube using the Centaur communications protocol. Our framework allows the programmer to break the hypercube up into one or more subcubes of arbitrary size (task parallelism). Each subcube runs a separate application using data parallelism and synchronous communications internal to the subcube. Communications between subcubes are performed with asynchronous messages. Subcubes can each define their own parameters and commands which drive their particular application. These are collected and organized by the Control Processor (CP) in order that the entire simulation can be driven from a single command-driven shell. This system allows several programmers to develop disjoint pieces of a large simulation in parallel and to then integrate them with little effort. Each programmer is, of course, also able to take advantage of the separate data and I/O processors on each hypercube node in order to overlap calculation and communication (on-board parallelism) as well as the pipelined floating point processor on each node (pipelined processor parallelism).

We show, as an example of the framework, a large space defense simulation. Functions (sensing, tracking, etc.) each comprise a subcube; functions are collected into defense platforms (satellites); and many platforms comprise the defense architecture. Software in the CP uses simple input to determine the node allocation to each function based on the desired defense architecture and number of platforms simulated in the hypercube. This allows many different architectures to be simulated. The set of simulated platforms, the results, and the messages between them are shown on color graphics displays. The methods used herein can be generalized to other simulations of a similar nature in a straightforward manner.

I. INTRODUCTION

Many applications in scientific computing cannot be solved with the homogeneous approach traditionally used with hypercube multicomputers. Solutions to inhomogeneous problems are required by such applications as electronic circuit simulations, war games, simulations of spacecraft systems, simulations of national or world economies, etc. Often such applications involve a degree of

time-dependence. That is, the character of the solution evolves with time. We call such applications inhomogeneous time-driven simulations and they are characterized by the following features: 1) They are composed of TASKS with various degrees of workload. 2) Tasks communicate with one another to perform the simulation. 3) Each task has a COMPUTATION CYCLE which is repeated many times duration the simulation. 4) Each cycle has four phases: a) reception of data from other tasks, b) processing of that data, c) communication of results to other tasks, d) advancing simulation time τ . 5) Different tasks may take different amounts of simulation time to perform their computation cycles, as well as taking different amounts of real time.

We have developed a general simulation framework for building and running such inhomogeneous time-driven simulations on hypercubes. The goals of our framework are to:

- 1. run tasks in parallel for maximum speed-up;
- load balance the processing power of the hypercube nodes so CPU-intensive tasks receive more CPU cycles than simple tasks;
- keep tasks distinct so they can be added, deleted, or replaced at will -- even at run time (however, we do not support the addition, deletion, or migration of tasks during the simulation);
- 4. allow multiple instances of each task to be simulated, the number of such instances also being determined at run time;
- 5. develop a communication system which can
 - a. determine which tasks communicate with each other and with what kind of data (at present we allow such dynamic configuration to occur only at run time, but we plan to support dynamic reconfiguration during the simulation in the near future),

- b. react to the inclusion of additional task instances as well as the non-inclusion of other tasks by developing an appropriate communication graph (again supported only at the beginning of the run at present),
- keep messages in proper simulation time and real time sequence, deliver them at the correct simulation time, and keep the system from deadlocking;
- allow the simulation to be controlled by the user from a single location, despite its multifaceted character.

In this paper we describe the methods by which we have implemented such a simulation framework and then discuss, as an example, a large space defense simulation -- "Simulation 88" -- which makes use of at least five different levels of parallelism available in the JPL/Caltech Mark IIIfp hypercube. We believe that Simulation 88 is one of the most sophisticated applications run on a hypercube to date.

II. THE GENERAL HYPERCUBE SIMULATION FRAMEWORK

A. Mixed Task and Data Parallelism Using the Centaur Operating System

Goals 1 - 4 are achieved in the following manner. Each task is decomposed onto a SUBCUBE of the hypercube (task parallelism). As well as possible, the number of nodes in each subcube is kept approximately proportional to the task workload per computation cycle divided by the desired simulation time per cycle (the throughput of the subcube). (Of course, the number of nodes in each subcube must be a power of 2.) Each subcube has a designated master node, the CORNER NODE, which communicates with corner nodes of other subcubes.

Within each task the computation is generally homogeneous. Therefore, algorithm speed-up is accomplished using data parallelism algorithms, i.e., the traditional homogeneous algorithms often proposed for hypercubes [1].

All communications, whether within or between subcubes, are handled by the CENTAUR OPERATING SYSTEM. [2] Within a subcube, the programmer uses fast synchronous communication subroutine calls (those from the so-called "crystalline operating system" or CrOS portion of Centaur). Between subcubes, specifically between corner nodes, and in communications with the outside world, the programmer uses asynchronous communication subroutine calls (those from the "Mercury" portion of Centaur). In Figure 1 we show a generic example of a 32-node hypercube decomposed into eight (8) separate subcubes, each of which is an instance of one of five distinct tasks.

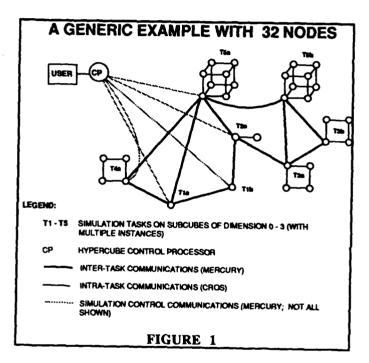
Our scheme for decomposing the hypercube into subcube tasks is described as below. Consider the following input parameters:

D -- dimension of full hypercube

Δt1i -- real time for task i to run one cycle on one hypercube node

Δτ; -- desired simulation time for one cycle of task i

n; -- number of instances desired for tasks i



To solve the decomposition problem, one must solve for the dimension of each task's subcube d_i, subject to the following constraints:

Each task's throughput must be load balanced as well as possible: $TP_1 \cong TP_2 \cong TP_3 \cong \ldots$, where $TP_i = 2^{d_i} \Delta \tau_i / \Delta t_{1i}$

Each task has at least one node: $d_i \ge 0$

Tasks must fill hypercube: $\sum_{i} n_{i} 2^{d}_{i} = 2^{D}$

These constraints are satisfied by the following algorithm:

- 1. Initialize all $d_i = 0$
- 2. Compute the throughput TPi of each subcube i

3. Choose the subcube j with the lowest throughput and compute the result of attempting to double the number of nodes for subcube j:

$$N_{test} = \sum_{i \neq j} n_i \ 2^{d_i} + n_j \ 2^{d_j+1}$$

- If N_{test} ≤ 2^D, replace dj ← dj + 1; else freeze dj, but continue searching for lowest throughput among other subcube tasks (i≠j)
- 5. If all d; have been frozen, exit the above loop, advancing to step 5; else to go step 2
- If the final N_{test} < 2^D, then there are spare nodes left, but there is no task which needs them or which can be doubled to fill them. Instead, fill them with a null task.

B. Constructing the Communications Graph (Between Corner Nodes)

Once the hypercube has been partitioned into subcubes, the set of communication links among the subcubes -- the communications graph -- must be specified. This graph, and the communications calls made during the simulation, are the key elements of our simulation framework. They ensure that the correct data are passed between tasks at the proper simulation times so that the tasks can continue to perform their computations without deadlock.

One important requirement of the communication system is that it must be able to build a graph given the number of tasks and their instances available in the hypercube at run time. It would be very cumbersome for the user if he were required to manually reconfigure the communication links every time he added or deleted a single task instance, or changed a task's throughput, thereby changing the number of nodes devoted to it. We have therefore implemented a general scheme where the user specifies GENERAL COMMUNICATION LINKS, which are valid under a wide variety of circumstances. These general links are then used by the framework to construct SPECIFIC COMMUNICATION LINKS at run time. The user need not know the number, size, or position in the hypercube of the subcube tasks in order to use this general scheme.

For each general link the user must specify:

The type of data being sent (a master list of allowed message types must be defined and be made part of the framework);

The sending task type (but not the instance nor the node number);

The receiving task type;

A COMMUNICATION RULE specifying to which of the several possible instances of sending tasks the receiving task should <u>LISTEN</u> for this message type.

Note that this "simulation mapping" process requires algorithms specific to the simulation being performed. It must be modified for each new simulation being developed.

The specification of "to whom to listen", rather than "to whom to send", is important. One can be derived from the other, but it is much easier to construct the latter from the former and insure that all tasks receive the data they need. In addition, by avoiding multiple sources of data for each type, this method insures that most (if not all) messages sent will be picked up and used by the receiving subcube. (This is useful as hypercube nodes have a finite amount of memory and cannot afford to leave a large number of unread messages.) There is also no need to create data arbitration algorithms for each communication reception to handle the case when more than one message of a given type arrives. However, this feature limits our framework to those simulations where tasks have only one source for any given data type. (Of course, additional data types can be defined to maintain the flexibility needed in most situations.)

At run time the general links are used to construct the specific communication links. A specific link is defined by 1) the sending subcube's corner node number and task type, 2) the receiving subcube's corner node number and task type, and 3) the message type. All links involving a single corner node are stored on that node in a lookup table. When a SEND of a certain message type is executed by a task, the intended receiving subcubes' type must also be specified in the communication call. (Only one call is needed to send to all subcubes of the same task type, but multiple sends must be performed if the same message type is intended for more than one task type.) The framework software then looks in the table for all links with the proper receiving task type and delivers the message to them. If no links satisfy the criteria, the call is ignored, but an error code is returned. Likewise, when a RECEIVE of a message type is executed, the software first checks the lookup table to see if the link has been defined.

The above scheme avoids deadlock in two cases: when a specific link is defined, but SEND and/or RECEIVE are not called; and when a link is not defined, and SEND or RECEIVE is called. Nevertheless, the tasks must be coded carefully as problems can still occur. Deadlock will occur if a link is defined and a RECEIVE is called by one task, but the sending task specified by the link has not called a SEND. Data overflow can occur if a link is defined and a SEND is called by one task, but the corresponding RECEIVE is not. The message queue on the receiving task then grows linearly with time.

C. The Synchronization of Tasks Using Message Passing, The Control of Simulation Time, And The Assurance of Task Parallelism

A message-passing system works properly only if the messages are sent and received at the proper times. Therefore, message passing cannot be considered without also considering the flow of time in the simulation and the method by which the tasks are synchronized. In our framework, synchronization is controlled by the message passing, just as it is in homogeneous applications, by forcing the receiving task to wait until it has received a message which satisfies certain criteria.

Each task has its own internal clock which advances simulation time in fixed increments of $\Delta \tau_i$. (Simulation time increments of different task types do not have to be the same.) Furthermore, in addition to the normal message header information which Centaur places on the message, our framework also time tags each message with the simulation time at which it is sent. A message sent by task j at simulation time τ_j and received by task i at time τ_i is accepted only if its time tag τ_i is in the interval

$$\tau_i - \alpha \Delta \tau_j \leq \tau_j < \tau_i + (1-\alpha) \Delta \tau_j$$
.

(α is a parameter which describes the type of message acceptance: α =1 denotes backward-biased, α =0 denotes forward-biased, and α =1/2 denotes time-centered acceptance.) Messages which are accepted are read from the queue but not discarded; only messages with tags $\tau_j < \tau_i - \alpha \Delta \tau_j$ are deleted. Note that the acceptance time interval is determined by the sender's simulation clock "tick" ($\Delta \tau_j$) and not the receiver's. This avoids deadlock regardless of whether the ratio $\Delta \tau_j / \Delta \tau_i$ is less than or greater than unity. If a message of the correct type, sending node, and time tag is not in the queue, the receiving task waits until one arrives. This synchronizes the tasks.

It is possible with such a synchronization scheme to force the tasks to execute in a sequential fashion and not in parallel! That is, it is possible that only one task at a time is performing any computations and that all the other tasks are waiting, especially if the communication graph has one or more closed loops embedded in it. This serial processing can be avoided if each task executes its operations in each cycle in a particular order:

- Send all messages; if there is no data to send, still send a null message (header);
- 2. Receive all messages from other tasks;
- 3. Perform CPU-intensive work
- 4. Advance simulation time by $\Delta \tau_i$ seconds

Sending all messages first "primes the pump" and allows other tasks to continue executing in parallel, especially when closed loops exist in the graph. Advancing the simulation time after the computation emulates the passing of simulation time during the computation portion of the cycle.

D. Centralized Simulation Control (The C3PO System)

Control of the execution of the simulation is provided by a program running in the Control Processor of the hypercube (see Figure 1): C3PO (Command and Parameter Processor for Program Organization). After the hypercube is partitioned, and before the communication links are set up, each subcube task defines a set of parameters and commands which control that task. (Typical parameters are names of initialization files, printing and plotting flags, etc.; typical commands are initialization, starting the execution of the cycles in each task, and commands which alter the simulation during execution such as shutdown.) The parameters and command names and types are sent up to C3PO in the CP where they are stored in a symbol table. All tasks then listen to C3PO for commands and continue to do so even while executing other commands.

A one-word command issued by the user at the C3PO prompt will execute a subroutine on all subcubes which recognize that command. In addition to commands, the C3PO interpreter also executes a C-like language. Parameter values may be altered at the C3PO level with assignment statements, C3PO functions, etc. Each command issued will use the latest values of the parameters.

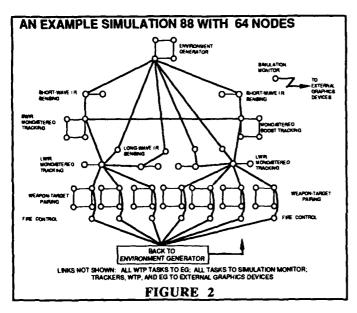
For sophisticated simulations the C3PO program itself can be a task with its own set of parameters and commands. This is most useful during the pre-initialization phase of a simulation when the partitioning of the hypercube into tasks is determined.

III. APPLICATION TO A COMPLEX STRATEGIC DEFENSE SIMULATION: SIMULATION 88

We have used the above framework to construct a detailed simulation of a strategic defense system. This simulation, called Simulation 88, is an emulation of a portion of a constellation of missile sensors, trackers, battle managers, and weapons platforms. Simulation 88 is composed of the following major tasks, each of which is a separate C program: SWIR (short-wave infrared) sensor; tracker of SWIR sensor data capable of stereo processing; LWIR (long-wave infrared) sensor; tracker of LWIR sensor data capable of stereo processing; a global engagement manager which allocates weapons in the arsenal based on ability to engage and the probability of kill; a fire control module which schedules weapon release and performs guidance; an environment generator which launches the threat, flies the

SDI platforms, and generally takes care of functions performed by the enemy or by nature; and a simulation monitor which doubles as the null task when not running on node 0 of the hypercube. In addition to being able to communicate with one another to assess and respond to the threat, most tasks can also open one or more windows on external color graphics workstations for viewing the simulation progress. The amount of C code running on these workstations is nearly equal to that running in the hypercube.

Simulation 88 has been designed and implemented in an unclassified environment. However, it is parameterized (through the use of C3PO parameters and initialization files) so that it can be run in a classified manner in the proper environment.



A run of Simulation 88 is uniquely determined by a configuration file which defines 1) which tasks are active, 2) how tasks are bundled together to form SDI platforms, 3) the total number of platforms of each type and their orbits, 4) how tasks communicate (the general links), 5) how many platforms of each type we wish to emulate in the hypercube (the rest are simulated in lower fidelity), and 6) how large a hypercube we wish to use for the simulation. All this information is parsed by the C3PO program before the hypercube is booted. After all executable code is downloaded into the hypercube, the specific platforms to be emulated are chosen from the constellations according to which ones can fight the battle best. All communication links between platforms are constructed, but only that subset which involves the chosen platforms is used for actual Centaur communications between tasks. Figure 2 shows the node allocation which results from a typical 64-node run. Increasing the cube dimension to seven (7), for example, would not change the number of modules but would change the numbers of nodes allocated to each.

Simulation 88 makes use of at least five different levels of parallelism:

<u>Multi-machine parallelism</u>: graphics processing and display occur in parallel with the hypercube simulation computations;

<u>Task parallelism</u> within the hypercube: the simulation is subdivided into subcubes; C3PO in the CP is also a task:

<u>Data parallelism</u> within each subcube of dimension $d_i > 0$: each task occupies 2^{d_i} nodes;

Intra-node parallelism: each task's code runs in the 68020/68882 processor or in the Weitek floating point processor; the Centaur communications is performed in parallel by a separate 68020 on each hypercube node;

<u>Pipelined parallelism</u>: some tasks execute their code in the Weitek floating point processor of the Mark IIIfp hypercube; this processor accomplishes parallelism on a machine instruction level.

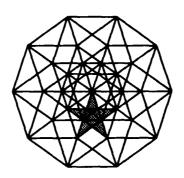
IV. ACKNOWLEDGEMENTS

The work described in this paper was carried out at the Jet Propulsion Laboratory, under contract with the National Aeronautics and Space Administration.

Simulation 88 itself is a much larger project than just the results reported herein. In addition to the above authors, the following contributed substantially to the individual pieces of software run on the various subcubes: environment generation -- R. Yeung; sensing / tracking -- T. Gottschalk, R. Yeung; weapon-target pairing / fire control -- D. Payne, E. Leaver, J. Steinman. The color graphics screens were developed by J. Lathrop, R. Iwashina, M. Pomerantz, and L. van Warren. We are also grateful for the considerable assistance given by members of the Hypercube system software team (R. Lee, C. Goodhart, L. Craymer, J. Crichton, N. Meshkaty, and B. Zimmerman) and of the hardware team (J. Peterson, M. Pniel, and D. Smith). The JPL Hypercube project is managed by D. Curkendall, the teams are managed by J. Fanselow, and the applications team is managed by D. Rogstad.

V. REFERENCES

- [1] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D., Solving Problems on Concurrent Processors, Prentice Hall, Englewood Cliffs, N.J., 1988.
- [2] Goodhart, C. and Lee, R. "Centaur: A Mixed Synchronous /Asynchronous Communication Protocol for the Mark III Hypercube", Fourth Conference on Hypercubes, Concurrent Computers, and Applications, Monterey CA, Mar 6-8, 1989.



The Fifth Distributed Memory Computing Conference

5: Path Planning and Navigation

Path planning on a distributed memory computer

Serge MIGUET Yves ROBERT

Laboratoire de l'Informatique du Parallélisme LIP-IMAG Ecole Normale Supérieure de Lyon 46 allée d'Italie 69364 Lyon Cedex 07, France

ABSTRACT

In this paper, we discuss the implementation of Bitz and Kung's path planning algorithm on a ring of generalpurpose processors. We show that Bitz and Kung's algorithm, originally designed for the Warp machine, is not efficient in this context, due to the intensive interprocessor communications that it requires. We design a modified version that performs much better. The new version updates a segment of k positions within a step and allocates blocks of r consecutive rows of the map to the processors in a wraparound fashion. Bitz and Kung's algorithm corresponds to the situation (k,r) = (1,1). We analytically determine the optimal values of the parameters (k,r) which minimize the parallel execution time as a function of the problem size n and of the number of processors p. The theoretical results are nicely corroborated by numerical experiments on a ring of 32 Transputers.

1. INTRODUCTION

Given a map on which each position is associated with a traversability cost, the path planning problem is to find a minimum-cost path from a source position to every other position in the map: look at the artificial example of figure 1. The altitudes of the points of this surface are proportional to their traversability costs. The top of the spiral-shaped hill has a high traversability cost, while the bottom of the valley is easier to go through. The source lies in the center of the spiral. We plot here a shortest path from the border of the domain to the source. We clearly see that the path hesitates between walking in the valley (long but easy), and crossing the hill (shorter in distance, but more costly).

Bitz and Kung [BK] have recently proposed a dynamic programming algorithm to solve the problem, and they have mapped this algorithm onto the linear systolic array in the Warp machine [AAG]. We show that Bitz and

Kung's algorithm is not efficient in the context of general purpose processors, due to the intensive communication scheme that it requires,

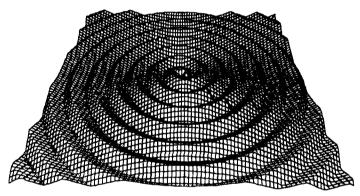


Figure 1: A shortest path on an artificial map.

2. PATH PLANNING ALGORITHM

A map M is an n x n grid of positions, for some positive integer n. The eight neighbors of a position p are indicated by the corresponding cardinal point in the compass (see figure 2).

NW	N	NE	
W	p	E	
sw	S	SE	

Figure 2: Labeling the eight neighbors of a position p

Each position p is associated with a non-negative realnumber tc(p) corresponding to the traversability cost of the position. Given a position p and a neighbor q of p, the edge (p,q) is weighted with a cost c(p,q) = (tc(p)+tc(q))/2 if $q \in \{N,S,W,E\}$ and c(p,q) = (tc(p)+tc(q))/2/2 otherwise: the $\sqrt{2}$ multiplier reflects the added traveling distance due

This work has been supported by the Research Project C3 of CNRS and by the ESPRIT Basic Research Action 3280 (NANA).

to the diagonal connection. Given a position, called the source, we want to compute the shortest path (or minimum-cost path) from it to every position in the map.

Bitz and Kung [BK] propose a dynamic programming algorithm to solve the problem. Initially, the best known cost f(p) for every position p in the map is assigned the value 0 at the source and ∞ at all other positions. The algorithm performs a succession of red and blue sweeps of the map.

2.1. RED SWEEP

The red sweep is a forward scan of the map M in the rowmajor ordering. During the red sweep, each position p is updated according to the red mask depicted in figure 3.

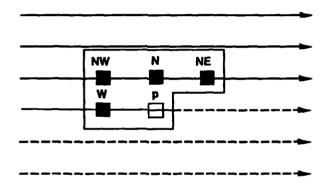


Figure 3: Red sweep and the associated mask

For the current position p of the sweeping, we update the best known cost f(p) if there exists a better path passing by one of the red neighbors of p. For instance if the best known cost f(W) of the west neighbor of p plus the cost c(W,p) of the edge from W to p is smaller than f(p), we update f(p) into f(p) := f(W) + c(W,p). In the general case, the update of f(p) is defined as

$$f(p) := min(f(p), f(W) + c(W,p), f(NW) + c(NW,p), f(N) + c(N,p), f(NE) + c(NE,p))$$

that is

(RS)
$$f(p) := min(f(p), f(W) + (tc(p)+tc(W))/2, f(NW) + (tc(p)+tc(NW))\sqrt{2}/2, f(N) + (tc(p)+tc(N))/2, f(NE)+(tc(p)+tc(NE))/2/2)$$

2.2. BLUE SWEEP

The blue sweep scans the map M in the reversed rowmajor ordering as shown in figure 4. For the current position p of the sweeping, the update of f(p) is defined similarly as for the blue sweep, but using the blue neighbors instead of the red ones:

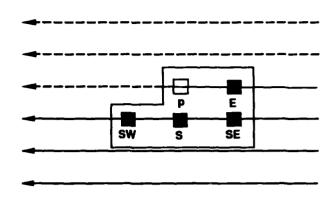


Figure 4: Blue sweep and the associated mask

$$f(p) := min(f(p), f(E) + c(E,p), f(SE) + c(SE,p), f(S) + c(S,p), f(SW) + c(SW,p))$$

that is
(BS)
$$f(p) := min(f(p), f(E) + (tc(p)+tc(E))/2, f(SE)+ (tc(p)+tc(SE))\sqrt{2}/2, f(S) + (tc(p)+tc(S))/2, f(SE)+(tc(p)+tc(SE))\sqrt{2}/2)$$

2.3. PATH PLANNING ALGORITHM

Given the initial values stated above, the red and blue sweeps are performed alternatively until no values are changed in one sweep. Let us color the edges of a path according to their directions: edges pointing to W, NW, N and NE directions are colored blue, whereas edges pointing to E, SE, S and SW directions are colored red. Then Bitz and Kung [BK] show that the number of required sweeps before all positions receive their final values is C or C+1, where C is the maximum number of color changes in a shortest path from the source to any other position. Hence in the worst case, the number of required sweeps can be as large as O(n²). However in practical situations, it is expected to be much smaller than n [BK].

In the following, we concentrate upon the parallel implementation of a single sweep (a red sweep) on a ring of processors.

3. PARALLEL IMPLEMENTATION

We briefly recall Bitz and Kung's solution for mapping the path planning algorithm onto the Warp. Such a solution is not suited to a ring of general-purpose processors, and we derive a modified version that performs much better.

3.1. BITZ AND KUNG'S MAPPING METHOD

We consider a ring of processors numbered from 0 to p-1. Each row of the map is assigned to a processor. Assume first that the number of processors p is equal to the problem size n. In this case processor i gets row i, 0≤i<n.

For the red sweep, immediately after processor i has computed the value of two positions, it will pass these values to processor i+1 to get it started. We summarize in figure 5 the time-steps at which each position is updated.



Figure 5: Time-steps for Bitz and Kung's parallel algorithm

At time 2i+j, Processor P_i operates as follows (wherever indices make sense):

- it receives position (i-1,j+1) from P_{i-1}
- it updates position (i,j)
- it sends position (i,j) to P_{i+1}

When p is smaller than n, partitioning techniques must be considered. Assume for the sake of simplicity that p divides n. Bitz and Kung propose to assign the rows of the map to the processors in a wraparound fashion: processor i gets rows j such that $j = i \mod p$. The wrap mapping is a widely used technique to well balance the workload among the processors [GH, MR, MV, Saa]. Now P_0 needs to receive computed values from P_{p-1} . Note that P_0 receives the first value (p-1,0) from P_{p-1} at time 2p-1. At time 2p, P_0 receives the second value (p-1,1) and updates position (p,0). Hence we do not want P_0 to finish the updating of row 0 before time 2p, otherwise it would stay idle for a while. This imply that $n \ge 2p$. If n > 2p, P_0 will simply store the values it receives from P_{p-1} until it starts the updating of its second row.

We see that the latency between the startup times of two adjacent processors is small (two time-steps). The major drawback of the algorithm is that is involves many short communications between the processors. For current distributed memory machines, the time to transfer L words between two adjacent processors can be modelized by β + L τ_c , and it turns out that β is significantly higher than τ_c ([GH, MV, Saa], see also the experiments reported in section 5). This renders the cost of small messages prohibitive.

We explain below how to modify Bitz and Kung's algorithm in order to decrease the communication overhead. We describe the new algorithm informally, and postpone its complexity analysis up to next section.

3.2. UPDATING A SEGMENT OF LENGTH K

The first way to decrease the communication overhead is to use longer messages. We use the same mapping strategy as before, but we update a segment of k consecutive positions at each step. The algorithm is illustrated figure 6. Note that k does not need to be a divisor of n. In figure 6, we let lo be the number of positions updated by P_0 at time 0: we choose $l_0 = k-1$ in our implementation, just as if Po had received k fictitious values before beginning (but lo can be any number between 1 and k-1). Each processor always updates k positions, except may be for the first and last updates: we start the update of the next row while finishing the update of the current row (see figure 6). The condition for Po not to finish its first row before receiving data from Pp-1 will be derived in the next section: we obtain the condition n ≥ (k+1) p.

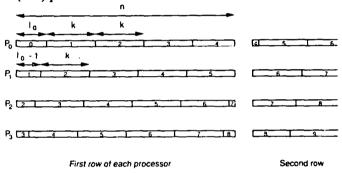


Figure 6: Updating a segment of length k

The number of data items communicated between two neighbor processors is exactly the same as before, but the larger k, the more efficiently the communications are performed. On the other hand, the larger k, the greater the latency between the startup times of two adjacent processors. We must be ready to find a compromise between the two contradictory exigences of mimimum startup delay (small k) and inexpensive communications (large k).

3.3. MOVING TO NEW MAPPING STRATEGIES

Another way to decrease the communication overhead is to communicate less data items between neighbor processors. We now consider more general allocation functions than the wrap mapping, and we assign blocks of r consecutive rows to the processors in a wraparound fashion [RTV]. For instance with r = 3, n = 36 and p = 4 we have the following repartition:

	P0	P1	P2	P3
Rows	0,1,2	3,4,5	6,7,8,	9,10,11
	12,13,14	15,16,17	18,19,20,	21,22,23
	24,25,26	27,28,29	30,31,32	33,34,35

Such a repartition is illustrated figure 7. Analytically, processor i gets rows j such that $i = \lfloor j/r \rfloor \mod p$, $0 \le j \le n-1$.

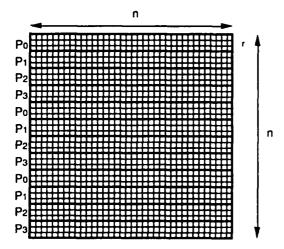


Figure 7: Block-r mapping, n = 36, p = 4, r = 3

The time-steps are depicted in figure 8. At each step except the first and last ones, all the processors update a parallelogram of r^*k positions. Just as before for r = 1, we start the update of the next block while finishing the update of the current block (see figure 8).

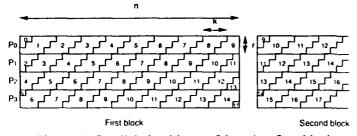


Figure 8: Parallel algorithm, n=36, p=4, r=3 and k=4

The condition that k and r must meet to keep all processors activated is the following: $n \ge p$ (r+k) (see next section). We show in figure 9 an example where this condition is not met: we see that P_0 is idle at time 4, because it has not received in time from P_3 the first positions of row 11.

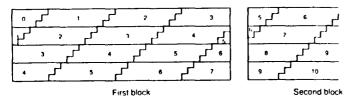


Figure 9: Parallel algorithm, n=36, p=4, r=3 and k=11

Now, the number of data items communicated between two neighbor processors is r times smaller than in Bitz and Kung's implementation, because the processors only need to exchange informations relative to the boundary rows of each block. Segments belonging to an internal row of a block do not require any inter-processor communication. We illustrate the communications between two neighbor processors in figure 10.

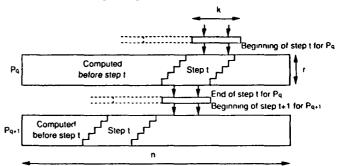


Figure 10: Communications between two processors

The price to pay for such a dramatic reduction of the communication volume is again an increase in the latency between the startup times of two adjacent processors. Hence the best value of r will result of a compromise, just as the best value of k.

In the next section, we perform a complexity analysis. Given n and p, we analytically determine the values of k and r that minimize the parallel execution time.

4. PERFORMANCE EVALUATION

In this section, we analyse the performances of the parallel algorithm described above. For the arithmetic, we let ta be the elemental time needed for updating a position during the sweep (formulae RS or BS). Since there are n^2 positions to update during a sweep, the sequential time for a problem of size n is $T_{seq} = n^2 \tau_a$.

4.1. MEMORY REQUIREMENT

The space requirement for the sequential algorithm is proportional to the size of the map, that is n² positions. For each position, we need to store a word for its current value and 8 words for the traversability cost of the 8 adjacent edges. Let us choose as a unit the memory requirements for a position. Given a single processor with a memory of size M, this implies that the maximal problem size that can be dealt with is $n_{max,1} = \sqrt{M}$. Consider now a ring of p processors. First of all, we have to determine the relationship between p and n. We have p memories of size M, so that we can solve in parallel a problem of size at most $n_{max,p} \equiv \sqrt{p} M$. Note that we neglect here any additional storage required by the parallel implementation, such as the need for communication buffers. In fact, the value n_{max,p} above is an upper bound.

As stated before, we consider an allocation by blocks of consecutive rows of size r in a wraparound fashion, where $1 \le r \le n/p$. For the sake of simplicity (without loss of generality), we assume that p^*r divides n, so that each processor holds the same number of rows in its local memory.

4.2. PARALLEL EXECUTION TIME

Even though the implementation is asynchronous, we can view the parallel algorithm as a succession of time-steps, where at each time-step each processor updates r segments of k positions. Within a time-step, processor P_i receives a message of length k from processor P_{i-1} , updates r^*k positions, and sends a message of length k to P_{i+1} (indices are taken mod p). Note that the emission is non-blocking, whereas the reception is. P_i does not wait for its emission to be completed before moving to the next step. As a consequence, the communication within a time-step has a cost equal to $\beta + k \tau_c$. The total time needed to perform a time-step is $\tau_{step} = \beta + k \tau_c + r k \tau_a$

To evaluate the total number of time-steps in the algorithm, we first compute the time-step at which a processor P_q , $0 \le q \le p-1$, initiates its computation. Recall that P_0 updates l_0 positions in its first row at time $t_0 = 0$. We see that P_1 updates $l_1 = (l_0-r)$ mod k positions in its first row at time $t_1 = 1 + \lceil (r-l_0) / k \rceil$, and more generally, that P_q updates l_q positions in its first row at time t_q , where

$$l_q = (l_0 - q r) \mod k,$$
 $t_q = q + \lceil (q r - l_0) / k \rceil$

Now, we derive easily the total number of time-steps T_p , since P_{p-1} is the last processor to end its computation. After updating its first parallelogram, P_{p-1} has still

$$T_p = t_{p-1} + \lceil (n^2/(p^*r) - l_q + r - 1) / k \rceil.$$
The parallel execution time of the algorithm is then
$$T// = \tau_{step} * T_p$$

This evaluation is valid only if the processors are not kept idle, waiting for some data they need from their predecessor. As explained in the previous section, this condition is equivalent to ensuring that P_0 has not finished the updating of its first block before receiving from P_{p-1} the data that it needs for its second block. P_0 performs its first reception at time t_p . At that time it has already updated $l_0 + k * t_{p-1}$ positions in the first row of its first block. The condition is that the sum of the remaining positions in this row plus the number of positions that it might update in the first row of the second block is greater than or equal to k, so that it can update a whole parallelogram at time t_p .

$$n - (l_0 + k*t_{p-1}) + l_p \ge k$$

After some algebra we get :
 $n \ge p (r+k)$

We retrieve the condition illustrated in figures 8 and 9.

Neglecting low order terms and ceiling functions, we obtain the following analytical evaluation for the parallel execution time $T_{//}$:

Proposition: Given a problem of size n and a ring of p processors, the parallel execution time $T_{//}$ for a block-r allocation, $1 \le r \le n/p$, using segemnts of length k, $1 \le k \le n/p - r$, is

$$T_{//} = \left(\beta + k \tau_c + r k \tau_a \right) \left[(p-1) \left(1 + \frac{r}{k} \right) + \frac{n^2}{p r k} \right]$$

Given n, p and r it is easy to find the value $k_{opt}(r)$ of k that miminizes the execution time $T_{//}$. We obtain the value

$$k_{opt}(r) = min(k_{max}(r), k_{//}(r))$$

where

$$k_{max}(r) = n/p - r$$

and

 $k_{//}$ is the optimal value obtained from the expression of $T_{//}$:

$$k_{f/f}(r) = \sqrt{\frac{\beta}{\tau_c + r \tau_a} \left(\frac{n^2}{p(p-1)r} + r\right)}$$

Given n, p and numerical values for the parameters β, τ_c, τ_a , it is easy to compute k_{opt} and to plug it into the expression of $T_{//}$ to determine the best value of r. We report numerical experiments in the next section.

5. NUMERICAL EXPERIMENTS

In this section,we report on numerical experiments on a ring of Inmos Transputers T414, using up to 32 processors. We use a FPS-T40 hypercube [GHS], which we configure as a ring. First of all we have to determine τ_a and τ_c .

Each update in (RS) or (BS) amounts to four additions, four comparisons, plus some conditional logic. We find that $\tau a = 75\text{e-}6$ seconds. For the communications, we obtain experimentally that the time to transfer L words between two adjacent processors is $\beta + L\tau_c$, with $\beta = 2\text{e-}3$ seconds and $\tau_c = 12.5\text{e-}6$ seconds.

The first thing we check is that the parallel execution time obeys our formulas. We fix n and p and let the segment size k vary, with various values of the block size r. We superimpose in figure 11 the experimental and theoretical curves (with the previous values of β , τ_c and τ_a) for the parallel execution time. There is a very good adequation between the curves.

We find experimentally the optimal values of r and k: r_{opt} = 6 and k_{opt} = 54. For these values we obtain $T_{//}$ = 10.36

seconds. These values are in good accordance with the theory: if we plug the values of n = 1920 and p = 32 in the formulas of the previous section, we obtain

$$k_{opt}(r) = \begin{cases} k_{max}(r) \text{ for } r \le 5\\ k_{//}(r) \text{ for } r \ge 6 \end{cases}$$

and $T_{//}$ is miminum for r = 6 and $k = k_{//}(r) = 54$. We obtain $T_{//} = 10.52$ seconds with the analytical expressions.

We point out that the execution time with (ropt, kopt) is divided by a factor of 23.8 as compared to Bitz and Kung's algorithm which corresponds to the values (r,k) = (1, 1) and for which the execution time is as high as 247 seconds.

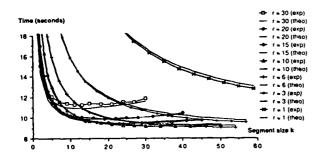


Figure 11: Parallel time as a function of k; n = 1920; p = 32

In figure 12, we plot the speedups that we obtain with 32 processors when solving a problem of size n = 1920. Note that these speedups are computed according Gustafson's recent proposal [Gus, CRT], in that they are normalized by the amount of arithmetic operations which they require (since it is impossible to solve such a large problem with a single processor). Using 32 processors, we report acceleration factors as high as 26.

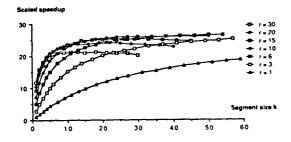


Figure 12: Scaled speedup as a function of k; n = 1920; p = 32

We finally show in figure 13 a 3D-plot of the efficiency e(r,k) of the algorithm to better visualise the influence of the parameters on the execution time. The surface we

show is the function e(r, k) for the following values of r and k: $1 \le r \le n/(2p)$, $1 \le k \le k_{max}(r)$. The optimal efficiency e = 0.81 is obtained for the highest point of this surface, with r = 6 and k = 54.

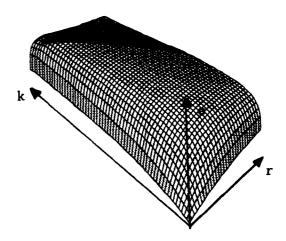


Figure 13: 3D-plot of the efficiency e(r,k), n = 1920, p = 32

6. CONCLUSION

In this paper, we have discussed the implementation of Bitz and Kung's path planning algorithm on a ring of general-purpose processors. We have designed a modified version that updates a segment of k positions within a step and allocates blocks of r consecutive rows of the map to the processors in a wraparound fashion. We have analytically determined the optimal values of the parameters (k,r) which minimize the parallel execution time as a function of the number of processors p and of the problem size n. The theoretical results are nicely corroborated by numerical experiments on a ring of 32 Transputers. We obtain a speedup of 23.8 over Bitz and Kung's algorithm.

7. REFERENCES

[AAG] M. ANNARATONE, E. ARNOULD, T. GROSS, H.T. KUNG, M. LAM, O. MENZILCIOGLU, J.A. WEBB, The Warp computer: architecture, implementation and performance, IEEE Trans. Computers 36, 12 (1987), 1523-1538

[BK] F. BITZ, H.T. KUNG, Path planning on the Warp computer: using a linear systolic array in dynamic programming, Intern. J. Computer Math. 25 (1988), 173-188

[CRT] M. COSNARD, Y. ROBERT, B. TOURANCHEAU, Evaluating speedups on distributed

memory architectures, Parallel Computing 10 (1989), 247-253

[GH] G.A.GEIST, M.T.HEATH, Matrix Factorization on a hypercube multiprocessor, Hypercube Multiprocessors 1986, M.T. Heath ed., SIA M (1986), 161-180

[GHS] J.L. GUSTAFSON, S. HAWKINSON, K. SCOTT, The architecture of a homogeneous vector supercomputer, in Proceedings of ICCP 86, IEEE Computer Science Press (1986), 649-652

[Gus] J.L. GUSTAFSON, Reevaluating Amdahl's law, Communications of the A.C.M. 31, 5 (1988), 532-533

[Hwa] K. HWANG, Advanced parallel processing with supercomputer architectures, Proceedings of the IEEE 75, 10 (1987), 1348-1379

[MR] S. MIGUET, Y. ROBERT, Dynamic programming on a ring of processors, Hypercube and Distributed Computers, F. André et J.P. Verjus eds., North Holland (1989), 19-33

[MV] O.A. MAC BRYAN, E.F. VAN DE VELDE, Hypercube algorithms and implementations, SIAM J. Sci. Stat. Comput. 8, 2 (1987), s227-s287

[RTV] Y. ROBERT, B. TOURANCHEAU, G. VILLARD, Data allocation strategies for the gauss and Jordan algorithm on a ring of processors, Information Processing Letters 31 (1989), 21-29

[Saa] Y. SAAD, Gaussian elimination on hypercubes, in Parallel Algorithms and Architectures, M. Cosnard et al. eds., North-Holland (1986), 5-18

LEARNING TO PLAN NEAR-OPTIMAL COLLISION-FREE PATHS

Alex W. Ho and Geoffrey C. Fox

Concurrent Computation Program 206-49, California Institute of Technology, Pasadena, CA 91125, USA

Abstract

A new approach to find a near-optimal collisionfree path is presented. The path planner is an implementation of the adaptive error back-propagation algorithm which learns to plan "good", if not optimal, collision-free paths from human-supervised training samples.

Path planning is formulated as a classification problem in which class labels are uniquely mapped onto the set of maneuverable actions of a robot or vehicle. A multi-scale representational scheme maps physical problem domains onto an arbitrarily chosen fixed size input layer of an error back-propagation network. The mapping does not only reduce the size of the computation domain, but also ensures applicability of a trained network over a wide range of problem sizes. Parallel implementation of the neural network path planner on hypercubes or Transputers based on Parasoft EXPRESS is simple and efficient. Simulation results of binary terrain navigation indicate that the planner performs effectively in unknown environment in the test cases.

Introduction

Robots have been successfully employed in very restricted, mechanical, and repetitive tasks such as to improve productivity and quality in assembly lines in automotive industry. Although it is not likely that man can construct even a near general-purpose robot in the foreseeable future given the current level of technology and advancement of science, the future generation of task-specific robot systems are expected to be more "autonomous" and "intelligent". These future robot systems would posses highly integrated capabilities of task-specific sensing – to gather relevant information of the environment and construct a limited world model of the physical surroundings,

goal-oriented planning – to achieve a high level specification of a goal by generating a sequence of robot actions in advance, motor control – to execution the planned sequence of actions step by step, and learning – to gain domain-specific knowledge from experience and response to unknown environment intelligently. In this paper, we confined our study to path planning for robot navigation.

The objective of developing autonomous robot navigation controller is to enable a robot to guide itself moving from one point of space to a destination without collision with the obstacles in its environment. The most basic form of a motion planning problem is the generalized mover's problem, which is also known as the Findpath or obstacle avoidance problem [1.] The goal is to find any collision-free path. For economic reasons, the path that a robot tracks should obey some constraints, which is usually in time and/or energy usages. For all practical purposes, the notion of planning a "good" path is of prime importance to any reasonable navigation controller.

There are many variants of the path planning problem. The task of planning an optimal path is achievable only for simple problems. Most of the time, the amount of computation required to obtain such a path could be costly. In many circumstances, optimal paths are not required. It is often more important to obtain a "good" (i.e., nearly but not precisely optimal) path quickly than to devote precious computational resources to find the exact solution. In fact the input data is often imprecise (e.g. the exact nature of the terrain is not known) and the notion of a precise optimal path undefined. Several new optimization techniques such as simulated annealing, neural networks, elastic networks and genetic algorithms have been devised for such approximate optimization problems [2-7.]

The use of a multi-layer feedforward neural network for path planning was first reported in [8], and some preliminary results on performance of such a trained network were discussed in [9.] In this paper, we will describe in detail the implementation of the neural approach used in [8,9] to the problem of planning a near-optimal, collision-free path for a single mobile robot moving in 2-dimensional binary terrains. Computational performance of the parallel algorithm on several distributed-memory MIMD processors like NCUBE-1, MEIKO Computing Surface (Transputer-based system), and iPSC-2 are compared. We will present algorithmic and implementation performances for cases of robot navigation on binary terrain.

Modeling The Navigation Terrains

We have chosen to apply our new approach to the simplest non-trivial path planning problem which is the navigation of a single vehicle in a plane with binary terrain. The terrain partitioning is a standard grid tessellation of the physical space of the problem domain which contains regions of random or structured obstacles. The remaining regions are robot traversable space. The discretized binary problem domain is represented as a 2-d matrix. A measure of the size of an instance of the path planning problem is the number of elements in the 2-d matrix. The higher the resolution of discretization, the bigger the problem size.

Although our technique can be extended to cover a more general path planning problem, the problem statement of our current study is stated as:

Given a discretized 2-d rectangular physical domain R, a distribution of binary obstacles D over R, the robot's current position $S \in R - D$, a high level specification of the target position $T \in R - D - S$, and a set of maneuverable actions or motor control constraints C governing the robot, find a near-optimal, collision-free path for the robot to move around in R from S to T.

Supervised Neural Network Approach

Artificial neural networks have been employed in a variety of applications, and were found most useful in the class of applications which a human operator can handle easily and efficiently. Obvious examples are in pattern classification and speech recognition. Another example is in playing chess. It has been a long-standing speculation that a good chess player recognizes abstracted patterns of the current board and commands a move with efficacy, while a chess playing computer program has to evaluate and search a huge game tree of legal moves and countermoves rooted from the current board, iteratively to a fixed depth.

In the same vein, the path planning problem can be transformed to a pattern classification problem in which class labels are uniquely mapped onto the set of maneuverable actions C of a robot, while each time instance of a scenario is mapped onto a 2-d binary pattern. We used a multi-layer perceptrons based on the adaptive error back-propagation algorithm [10] as the pattern classifier for the transformed path planning problem.

The back-propagation model has been widely used for pattern recognition tasks. The architecture of such a neural net model has an input, output, and intermediate layers. All inter-layers are fully connected. Unlike the Hopfield model which has recurrent connections[4,] the perceptrons model does not provide a feedback mechanism for neuronal activation to propagate. A set of training sample pairs which carries some form of relevant information about the classification problem at hand is used to train the multi-layer network. Iterative synaptic weight adaptation occurs following the back-propagation algorithm as the error signal at the output layer is propagated backward and filtered by the same set of synaptic connections for forward propagation.

The Issue of Representation

The success of the back-propagation model on pattern recognition problems relies heavily on the choice of the pre-processing operations. The choice of pre-processing operations for raw pattern data determine the selective pruning and encoding of information. The representations that emerge from these operations impose constraints on subsequent processing by the back-propagation neural model.

A reasonable pre-processing scheme should be one which reduces and encodes raw patterns into some form of standard representations. We adopted a non-linear, multi-scale sampling strategy which mapped a physical problem domain onto an arbitrarily chosen fixed size input layer of an error back-propagation network. The multi-scale representation of patterns is a natural consequence of the sampling strategy used.

The sampling strategy involves using high resolution neurons to encode terrain information close to a robot, and progressively coarser neurons away from a robot by increasing the sampling interval. The physical problem domain is separated, in a fuzzy sense, into a near field and a far field. Near field information which is encoded in high resolution neurons is used to generate immediate action (corresponds to local planning,) while far field information which is encoded in the coarse neurons is used for global planning.

The only traffic regulation imposed on robot motion is that every move must be collision-free. In this study, a robot was restricted to move in one of the five admissible directions in a 2-d rectangular problem domain, and the cost to move in any of the five admissible directions is the same. Our formulation of robot maneuverable motions could be extended to eight directions to include the eight nearest neighbor in the case of a 2-d grid tessellation. Figure 1 shows the five admissible moves of a robot. Since the cost of selecting to move in any of these five directions are the same, an optimal collision-free path would be one which minimizes the number of moves required to get from a source point to a target position without violating the traffic regulation.

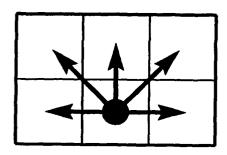


Figure 1: Vehicle is constrainted to move in one of the five directions. All moves have the same cost.

The five admissible moves are mapped one to one onto five grandmother cells at the output layer

of a back-propagation model. The activation values of the five grandmother cells indicate how good it is to move a robot in each of the five corresponding directions (see Fig. 2.)

ACTIVATION OF OUTPUT NEURON INDICATES SCALE OF GOODNESS



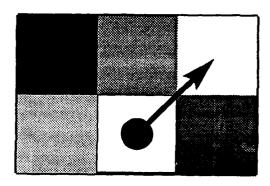


Figure 2: A robot moves in the direction which corresponds to the highest activation voltage.

We have arbitrarily chosen to use four different scales and nine general directions (East, NEE, NE, NNE, North, NNW, NW, NWW, West) to represent each scenario. This sampling strategy divides the problem domain into 36 regions. At the input layer, one neuron is needed to encode information for each region which is actually one combination of direction and scale. All together, 36 neurons are needed. An intermediate layer with 20 neurons was used. The number of neurons in the intermediate layer was arbitrarily chosen to achieve a fan-in architecture.

The activation value for each input neuron is a function of a porosity index which is a measure of the traversability of the corresponding region. The porosity index is taken as the compliment of the density of obstacles. The input neuron activation is computed by using Eq. (1)

$$V_{\eta_i} = f(1 - \rho, \gamma_i) + \delta(target, \gamma_i)$$
 (1)

where $V_{\eta_i} \in [-1.0, 1.0]$ is the activation value for neuron η_i in the region γ_i , ρ is the density of obstacles in γ_i , and δ is the Kronecker delta which equals to 1 if the target position is in γ_i , and zero otherwise.

Parallel Back-Propagation

The back-propagation algorithm is an effective training algorithm for the feed-forward multi-layer perceptron model. It is a generalization of the least mean square algorithm or the delta rule. Back-propagation uses a gradient descent technique to minimize a quadratic error function which is defined as the mean square differences between the pair of actual network output vector and its associated target vector for the set of training samples.

Let us define the following:

- w_{ij} is the connection weight between the jth neuron in the current layer and the ith neuron in the immediate lower layer,
- θ_j is the internal threshold of the jth neuron in the current layer,
- x_i is the ith continuous-valued input from the input layer or the layer underneath the current layer.
- x'_j is the output of the j^{th} neuron in the current layer,
- y_j is the actual model output of the jth neuron in the output layer,
- and d_j is the desired or target output of the jth neuron in the output layer.

The model is trained by initially assigning small random weights to the synaptic connections and small random thresholds to the artificial neurons. The neuronal outputs from each layer are then computed by

$$x'_{j} = f(\sum_{i=0,\ldots,N-1} w_{ij}x_{i} - \theta_{j}) \qquad (2)$$

where N is the number of neurons in the layer below the current layer. If the current layer is the physical output layer,

$$y_j = x'_j. (3)$$

The neuron activation function f has to be non-decreasing, continuously differentiable. Usually, the sigmoid logistic function

$$f(\zeta) = \frac{1}{1 + e^{-(\zeta)}} \tag{4}$$

is used for this purpose.

Given the measure of the error on any pattern in the training set as

$$E = \sum_{j} (d_j - y_j)^2 \qquad (5)$$

and the neuron activation function f as described in Eq. (4), adaptive correction of the connection weights in the direction of $-\partial E/\partial w$ corresponds to performing a steepest descent search in the weight space to minimize error. Synaptic weight adaptation follows Eq. (6)

$$\Delta w_{ij}(t+1) = \eta \delta f(\zeta) + \alpha \Delta w_{ij}(t) \tag{6}$$

where η is the learning rate, α is the momentum term which determines how much is remembered about the previous iteration, and δ is the filtered error signal.

Similarly, the internal thresholds θ_j are corrected adaptively in the threshold space. The partial derivatives $-\partial E/\partial w$ and $-\partial E/\partial \theta$ are computed by propagating error signals from the output layer back to the lower layers through the net, which motivates the name "back-propagation".

Parallel implementation of the back-propagation model for Chinese character recognition on hypercubes based on a character decomposition technique using bitmap masks has been discussed by [11.] Our current implementation of the path planner is based on a distribution of the set of training patterns over the number of processors of a hypercube. Essentially, each processor of an allocated hypercube or Transputer-based concurrent processor is responsible for only a small subset of the set of training samples. Using Parasoft EXPRESS as the communication software the same code runs on NCUBE-1, iPSC-2, and on Meiko Computing Surface which is a Transputer-based system.

The training set we used consists of 184 patterns, which contains knowledge of 184 scenarios of human-supervised optimal and collision-free moves in binary terrain navigation. The choice of using 184 patterns is arbitrary. Our first goal is to teach the navigator enough basic knowledge upon which it can generalize, not just memorize, to cope with most situations.

Learning Histories of A Back-Propagation Path Planner

Besides the issue of pattern representation, parameter tuning is a major concern for the back-propagation model to converge fast, or converge at all. Convergence of the model depends on the initial configuration of the network, the choice of the learning rate η , and the momentum term α .

The learning histories for four different initial configurations with fixed learning rate $\eta=0.10$, and momentum $\alpha=0.80$ are shown in Fig. 3. Average error is defined as the average of the total quadratic error per pattern per output neuron. The backpropagation path planner converged for these four cases.

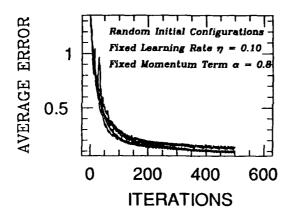


Figure 3: The learning histories of a back-propagation path planner for four different initial configurations.

To study the effect of the momentum term α on the convergence of the model, we used a fixed initial configuration and set $\eta=0.10$. Figure 4 shows the learning histories for $\alpha=0.2,\,0.4,\,0.6,\,$ and 0.8. The behavior of the model in its learning phase were very similar for the four different values of α . All four cases converged roughly at the same rate because they had the same learning rate.

More dramatic effects were observed, as expected, for the cases of using different learning rates η , and fixing the initial configuration and the value of

α. The learning histories for seven different learning rates are displayed in Fig. 5.

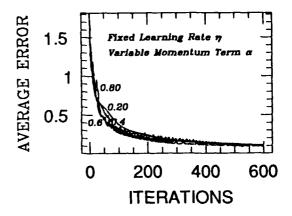


Figure 4: The learning histories of a back-propagation path planner using four different values for the momentum term.

In general, the bigger the learning rate the faster the convergence. However, when the learning rate is set to a value that is "too big", it leads to big oscillations and instability. Small learning rates usually lead to smooth but slow learning.

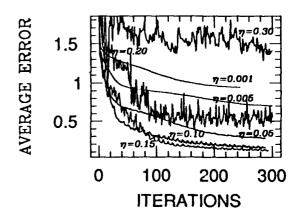


Figure 5: The learning histories of a back-propagation path planner for seven different learning rates.

Performance of A Trained Back-Propagation Path Planner

We tested the performance of a trained back-propagation neural path planner by submitting to it unlearned scenarios in the form of 47×24 as well as 105×53 discretized map. Several instances which include random and structured obstacles are shown in Fig. 6 to 10.

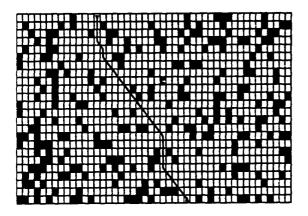


Figure 6: A 47 × 24 binary terrain with randomly distributed obstacles. Triangle indicates starting location, and an inverted triangle indicates target position. An optimal collision-free path was planned.

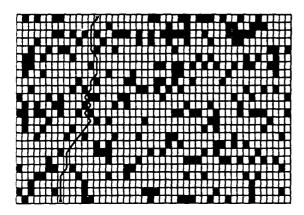


Figure 7: A 47 × 24 binary terrain with randomly distributed obstacles. An optimal collision-free path was planned.

Since an optimal path in this study is one which minimizes the number of collision-free moves needed

to get from a source to a target position within the problem domain, the paths displayed in Fig. 6, 7, 9, and 10 are optimal. However, the planned path is near-optimal in Fig. 8.

Performance of Parallel Implementation

We used up to 64-node NCUBE-1, iPSC-2, and 16-node Meiko for our simulations. A training set of 184 patterns becomes a small problem for the case of 64 processors because each processor is then responsible for performing computations for at most 3 patterns. Figure 11 shows the timing result for running one iteration of the back-propagation path planner. The reported time is normalized to that needed to run one iteration of the same planner on a 20 MHz SUN4/60 SPARCstation 1. For the one processor case, a Meiko Transputer node was the fastest, achieving a performance close to that of a SUN4/60. The efficiencies of the same program on the three different concurrent processors are shown in Fig. 12. Although the efficiency for the simulations performed on an NCUBE-1 seems to be better than on an iPSC-2 or a Meiko Transpoter systems, this result should be taken with care. "even though exactly the same EXPRESS program was used for simulations, there were differences in the hard-wired configuration of the three computer systems, and in the implementation of EXPRESS.

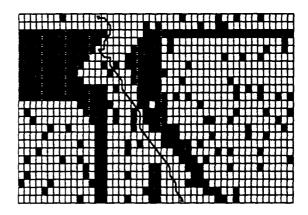


Figure 8: A 47×24 binary terrain with structured obstacles. The planned path is near-optimal.

There is one simple explanation for the poor efficiency of the simulations on a Meiko Computing Surface. EXPRESS communications are best suited for hypercube connectivity concurrent processors. At

the time of the simulations, the Transputer system was hard-wired as a 2-d torus instead of a hypercube; thus, resulting in poor performance.

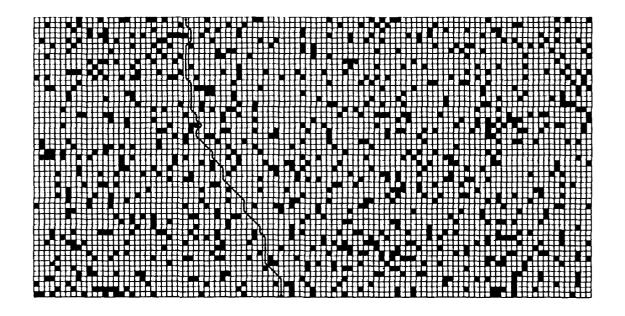


Figure 9: A 105×54 binary terrain with randomly distributed obstacles. An optimal collision-free path was planned.

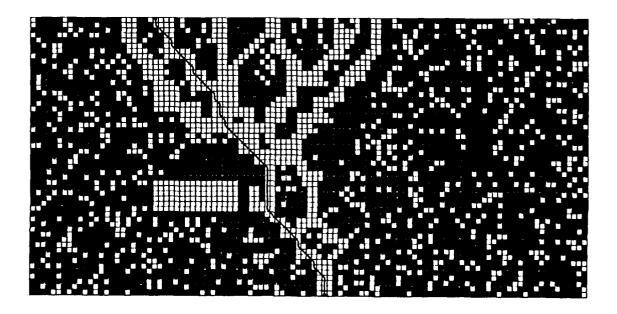


Figure 10: A 105×54 binary terrain with structured obstacles. An optimal collision-free path was planned.

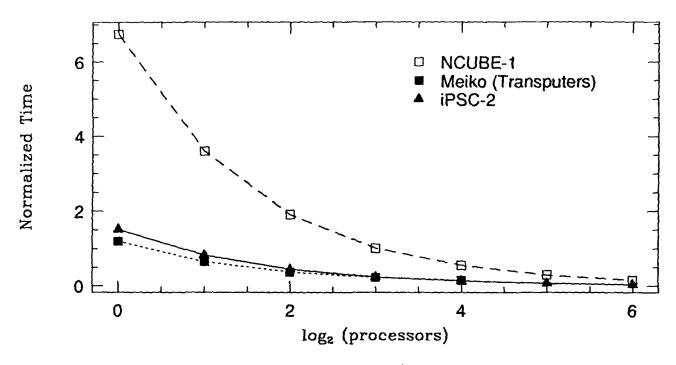


Figure 11: Timing result normalized to the time required to run one epoch on a 20 MHz SUN4/60 SPARCstation 1.

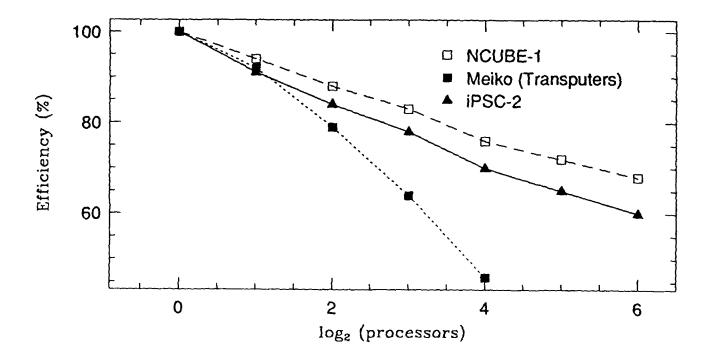


Figure 12: Efficiency plotted as a function of the dimension of a hypercube.

EXPRESS was optimally implemented on the NCUBE-1. As for the iPSC-2, EXPRESS was implemented on top of the native operating system (NX). This extra layer in between EXPRESS and the hardware is expected to incur inefficiency.

Conclusions

Simulation results indicate that a trained back-propagation path planner possesses satisfactory capability of planning near-optimal, collision-free paths in binary terrains with random or structured obstacles. The multi-scale mapping scheme does not only reduce the size of the computational domain and encode sufficient information to carry out the planning task, but also ensures applicability of the trained network on a wide range of problem sizes.

The advantages of this new approach of transforming a path planning problem to one in pattern classification by neural networks are:

- External homing strategy is not required.
- No explicit heuristic is used for shortest path.
- No need to decompose the problem domain into configuration space and free space.

The homing strategy, and the notions of optimality and obstacles avoidance are all encapsulated into the training patterns as task-specific knowledge from a human teacher.

Acknowledgement

This study is based on research work supported by the Joint Tactical Fusion Program Manager.

References

- [1] Schwartz, J.T. and Sharir, M., "A Survey of Motion Planning and Related Geometric Algorithms," Artificial Intelligence 37, 157-169 (1988.)
- [2] Simic, P., "Statistical Mechanics as the Underlying Theory of 'Elastic' and 'Neural' Optimizers," NETWORK: Computation in Neural Systems 1, 1-15 (1990), Technical Report CALT-68-1556, C³P-787, California Institute of Technology, May 1989.
- [3] Geoffrey Fox, Wojtek Furmanski, Alex IIo, Jeff Koller, Petar Simic, and Issac Wong, "Neural

- Networks and Dynamic Complex Systems", contribution to the 1989 SCS Eastern Conference, Tampa, Florida, (March 1989,) Technical Report C³P-695, California Institute of Technology, 1988.
- [4] Hopfield, J. J. and Tank, D., "Neural Computation of Decisions in Optimization Problems," Biol. Cybernetics 52, 141-152 (1985.)
- [5] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by Simulated Annealing," Science 220, 671 (1983.)
- [6] Fox, Geoffrey C. and Gurewitz, Eitan and Wong, Yiu-fai, "A Neural Network Approach to Multivehicle Navigation", contribution to the 1989 SPIE Conference, Philadelphia, Pennsylvania, (Nov. 1989,) Technical Report C³P-833, California Institute of Technology, 1989.
- [7] Wong, Issac and Fox, Geoffrey C., "Use of neural networks for path planning," Technical Report C³P-784, California Institute of Technology, May 1989.
- [8] Ho, Alex W., "A Back-propagation Navigation Controller for Land and Space Vehicles," Technical Report C³P-735, California Institute of Technology, April 1989.
- [9] Ho, Alex W. and Fox, Geoffrey C, "Parallel Neural Net Planner on Hypercube and Transputer," in Parallel Processing in Neural Systems and Computers, eds. R. Eckmiller et. al., Elsevier, 421-426, (1990.) Technical Report C³P-848, California Institute of Technology, 1989.
- [10] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning Internal Representation by Error Propagation" in D.E. Rumelhart & J.L. McClelland (Eds.), "Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations." MIT Press, 318-364 (1986.)
- [11] Ho, Alex W. and Furmanski, W., "Pattern Recognition using Neural Networks in Hypercubes," in the Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. 2, (ed.) Geoffrey C. Fox, ACM Press, New York, 1011-1021 (1988), Technical Report C³P-528, California Institute of Technology, 1988.

Parallel Algorithms for One and Two-Vehicle Navigation

Eitan Gurewitz*, Geoffrey Fox, Yiu-fai Wong

Caltech Concurrent Computation Program
California Institute of Technology
Mail Code 206-49, Pasadena, CA 91125

Abstract

A two vehicle navigator on a descrete space is analyzed. The concept of linking time maps as source to optimal path planning is discussed. The rules for constructing these maps are given in a cellular automata mode. The implementation of these rules on a parallel computer is presented.

1. Introduction.

In this study navigation means determination of a path on a navigation surface [NS] from an origin point to a destination point. A cost function is defined on the NS, measures the cost of traveling a length segment. The cost can be: time, length, hazard of traveling the segment etc. An optimal path on the NS is a path along which the integration of the cost function from origin to destination is minimum. The objective of a navigator is to find the optimal path under the constraints set by the NS. The problem of an optimal path for a single vehicle on a continuous surface [1] as well as a discrete surfaces [2] were solved. This study analyses the two vehicle navigator and presents the linking time maps as a tool to deal with these problems.

A discrete solution for navigation on a continuous space requires mapping of the space into a finite graph. This is done by choosing a finite number of points $\{v_i\}$ on the surface as the nodes of the graph. Each node is connected by an edge to all the nodes which can be reached, without traversing another node. The set of all the nodes $\{v_i\}$ having a common edge with v_i is the set of v_i nearest neighbors [nn(i)]. The value w_{ij} of the cost of traveling along the directed edge $[v_i, v_j]$ is assigned to this directed edge. This procedure maps the surface onto a directed graph, Fig. 1. Mapping the NS onto a directed graph transfered the search for an optimal path to the the search for an optimal path on a directed graph. This search is solved by a dynamic programing approach [3], where a "signal" is initialized at a source point and propagates from a node to all its nn along the edges joining them. The time the signal travels along an edge is the weight of the directed edge. Every node v_i records the first time t_i it was hit by the signal. The graph in which all the nodes v_i have their correct time values t_i is called: the *linking time map* [LTM] with respect to the generating node.

In fact the linking time t_i at the node v_i is the cost of an optimal path from the source to this node and it depends only on the weights of the edges and the generating node. The linking times t_i and t_j of two sequential nodes v_i and v_j on an optimal path, where v_i proceeds v_j are related by:

$$t_j = t_i + w_{ij} \tag{*}$$

An optimal path from the origin to any point on the graph is traced from that point back to the origin. Every step is from a node $v_j(t_j)$ to a node $v_i(t_i)$ where t_i and t_j satisfy (*). Tracing back ensures that one stays on an optimal path initialized at the origin.

Let us call the traveling object a vehicle and consider the case of two vehicles traveling on the same NS. If the path of each vehicle introduces restrictions to the path of the other vehicle (e.g. collision avoidance) then a search for an optimal solution is much more complicated.

The layout of this study is: In section 2 we discuss navigation of an autonomous vehicle on an NS which is updated while traveling. In section 3 we intoduce time and deal with conflicts between vehicles. The resolution of a conflict by imposing a delay on a vehicle is discussed and the paths solving a two vehicle navigator are analysed. Section 4 outlines briefly the algorithm for two vehicle navigator. Section 5 presents the cellular automata rules for constructing linking maps. Section 6 deals with the

actual parallel implementations of the construction of linking maps, and section 7 presents the simulation results.

2. Autonomous vehicle in uncertain environment

An autonomous vehicle in an uncertain environment start with an estimate of the edges weights. The estimate reflects the prior knowledge or model it has for the terrain to be transversed. The estimate is improved as more information is obtained. The vehicle knows its position and destination and at each instance of time the vehicle is doing the following: 1) updates the database of the weights $\{w_{ij}\}$. 2) Based on the updated data it determines the optimal path from its current position to the destination. 3) Moves on the chosen optimal path. 4) Collects data.

Updated weights $\{w_{ij}\}$ change the LTM, but a change in the linking time of a node may effect the linking times of only part of the other nodes. In section 5 we show how to update the LTM in a cellular automata fashion, based on local decisions of each node.

The navigator for an autonomous vehicle is based on the reversed linking time map [RLTM]. The construction of the RLTM is similar to the construction of the LTM. Except that in constructing the RLTM the signal is initializing at the destination point and propagates from v_i to v_j with traveling time of w_{ji} . The path is traced from the vehicle position toward the destination, from v_i with reversed linking time θ_i to its nearest neighbour v_j with reversed linking time θ_j which satisfies:

$$\theta_j = \theta_i - w_{ij}$$

Whenever the vehicle gets new information it updates the $\{w_{ij}\}$ database and its RLTM, and determines an optimal path, Fig. 2.

Navigation in Space-Time, and non conflicting paths for two vehicles.

Assume that the cost function is time, i.e. the weights $\{w_{ij}\}$ are the time of travel along the corresponding edges. Then a navigator for two vehicles aims to find two paths, one for each vehicle, which yield the minimum time of travel.

Assuming the two vehicles start at the same time, then time of travel is the time it takes until both of them have arrived. This optimum is restricted to non conflicting paths.

A conflict between two paths occurs when the two vehicles are at the same site at the same time. The set of points on the graph edges is partitioned into sites as follows. Each point is associated to the nearest of the two nodes terminating the edge. A conflict can occur either: a) inside this site or b) a swap conflict on the boundary between two sites. In the second case the vehicles are going in opposite directions. Let v_i^0, v_j^0, v_k^0 and v_i^1, v_j^1, v_k^1 , be three sequential nodes on the paths of vehicle 0 and vehicle 1 respectively. The node v_j is on the two paths. A conflict of type (a) at v_j occur if and only if

$$t_{j}^{1} - \frac{w_{i'j}}{2} < t_{k}^{0} - \frac{w_{jk}}{2}$$

and

$$t_{j}^{0}-\frac{w_{ij}}{2}< t_{k'}^{1}-\frac{w_{jk'}}{2}$$

A conflict of type (b) at the boundary between v_j and v_k occur if and only if:

$$i' = k$$

and

$$t_k^0 - \frac{w_{jk}}{2} = t_j^1 - \frac{w_{i'j}}{2}$$

To resolve the conflict at v_j one vehicle cannot enter into the site until the other clears the site of v_i . In the graph representation this is done by imposing a *delay* w at v_i on either one of the two vehicles:

$$w = t_{k'}^1 - \frac{w_{jk'}}{2} - t_j^0 + \frac{w_{ij}}{2}$$

on vehicle 0, or

$$w = t_k^0 - \frac{w_{jk}}{2} - t_j^1 + \frac{w_{i'j}}{2}$$

on vehicle 1. Imposing a delay w at v_i on vehicle k means that t_i^k is set to $t_i^k = t_i^k + w$ and LTM^k is accordingly updated. imposing a delay on a vehicle and updating its LTM preserves the characteristic of the LTM to yield, by the tracing back procedure, the optimal paths under the imposed restriction.

If the optimal paths of vehicles 0 and 1 have more than one conflicting nodes then: 1) their path segments from the first to the last conflict have exactly the same time of travel. 2) On these equivalent segments they are traveling in the same direction. When the two paths have

more than one conflict, the resolution of each conflict requires the minimal delay given above. Therefore, imposing the maximal delay of these waits on the first node of conflict resolves all the conflicts between these two paths. However, the path with the delay on it may not be an optimal path anymore.

Consider the case where an optimal path of one vehicle conflicts, at v_i with the optimal path of the other vehicle. Assume that the required delay at v_i was imposed on one of the vehicles, its LTM was updated and a new optimal path was traced. Then one of the following will occur:

- The new path does not conflict with the path of the other vehicle, and the are candidates for an optimal solution.
- The new path conflicts with the path of the other vehicle, but it does not pass through v_i.
- 3. The new path passes through v_i and it conflicts with the path of the other vehicle. In this case the new conflict is a swap conflict at the boundary between v_i and its proceeding node on the other vehicle path.

In an optimal solution of the two vehicle navigator there cannot be an instant when the two vehicles are waiting. Therefore, the paths solving this problem can be of three types:

- 1. Neither of the vehicles waits.
- 2. One of the vehicles waits.
- 3. The two vehicles have to wait. The last case happens resolving a swap conflict when vehicle k has to wait for vehicle l to step aside letting k to path and then looping or detouring.

Let us extend the NS by adding to it the time dimension Fig. 3. The graph $\{v_i, e_{i,j}(w_{i,j})\}$ on the navigation plane is the projection of the extended graph on the t=0 plane. The linking time value t_i of a node v_i is its t-coordinate in the extended space. The linking times t_i and t_j of two sequential nodes, v_i and v_j , on a legal path in the extended space are restricted to the condition (1). In the extended graph delay means that two sequential nodes on a path have the same NS coordinates but different time coordinate. A loop means that two

non sequential nodes on the path have the same NS coordinates but different time coordinate. A detour means that two sequential nodes on the path do not obey the path rule, i.e. $\theta_i + w_{ji} > \theta_j$.

The space-time representation of the paths depicts the difference between this problem and the K-disjoint[4] problem. In this problem we do not know the t-coordinates of the destination points. These points are subjected to the searching process.

The complexity of a search for an optimal solution for multiple vehicles grows fast with the number of vehicles. For this reason, other suboptimal methods are investigated, such as neural networks [5,6].

4. Algorithm for the two vehicle navigator.

The algorithm for the two vehicle navigator is based on the concepts discussed in the previous section using the cellular automata rules of the next section. The idea is to hold LTM and RLTM for each vehicle and to update them wenever a restriction is set. The need for a RLTM arises whenever a swap conflict occur, and a search for a loop or a detour is regarded.

As was already stated: the two vehicles cannot wait at the same time, and a solution which imposes delays on the two vehicles is obtained only when one of the paths is a loop or a detour. Therefore, the algorithm finds two separate solutions. A solution when the delays are imposed on vehicle 1 only and a solution where the delays are imposed only on vehicle 0. When imposing a delay to resolve a swap conflict the algorithm checks for loop or a detour. The best of these solutions is the optimal solution. In practice the algorithm will not construct those two solutions, but to minimize computations, it will prune the search by always adjusting the path of the vehicle with the sorter time.

On a binary speed NS the speed of the vehicle at each point is either 1 or 0. The two vehicle navigation problem on this NS is much easier as the rules get simpler form. on this NS a conflict of type (a) is at the node itself and it needs w = 1 to be resolved. The swap conflict (of type (b)) needs w=2 to be resolved. Fig. 4

presents the two vehicle navigator solution for a conflict imposing NS.

5. Cellular automata instruction for the navigation algorithm

- Rule 1: The linking time of the generating point is always $t_0 = 0$.
- Rule 2: The linking time t_i of every node $v_i i \neq o$ is:

$$t_i = Min\{t_i, t_j + w_{ii} | \forall j \in nn(i)\}$$

,where nn(i) are all vi nearest neighbors.

Rule 2': The reversed linking time θ_i of every node $v_i i \neq o$ is:

$$\theta_i = Min\{\theta_i, \theta_i + w_{ii} | \forall j \in nn(i)\}$$

- Rule 3: If a node other than the generator does not have a source, it set its linking time to infinity. Namely, if $i \neq 0$, and $t_i > t_j \forall j \in nn(i)$ then $t_i = \infty$.
- Rule 3': If $i \neq o$, and $\theta_i > \theta_j \forall j \in nn(i)$ then $\theta_i = \infty$.

Algorithm for constructing the LTM or RLTM:

- 1. Initialize the linking times of all the nodes to "infinity".
- 2. Set the generator linking time to 0.
- 3. Apply rule 2 or 2'.
- 4. When there is not a node which update its value the LTM or RLTM is done.

Algorithm for updating the LTM or RLTM where a delay W is imposed on v_i :

- 1. Set $t_i = t_i + W / \theta_i = \theta_i + W$
- 2. Apply rule 3 / 3'.
- 3. Apply rule 2 / 2'.

6. Parallel implementation of the timelinking map

The cellular automata mode of constructing the LTM is asynchronous but the linking

process has a propagating nature. The wave front of the propagating linking signal depend on the data and the location of the generating node. Therefore, the scattered decomposition [7] would be the most appropriate decomposition approach. The mapping in this approach is as follows: The NS is tessellated into $N_x x N_y$ congruent templet. Each templet is tessellated again to K equal tiles, where K is the number of processors. Each processor is assigned to the same tile of the templet over all the templets, Fig. 5. As the computational graph in our case is very irregular and time dependent, the scattered decomposition will hopefully balance the work done in each processor.

As the information propagates from a node to its neighbors the smaller the tiles in each templet are the greater the number of nodes propagating the correct linking time is. On the other hand the smaller the tile is the greater the number of nodes on the boundary is. Therefore, for given number of processors and dimension of the descrete NS there is an optimal size of tile. The bigger the number of processors is the smaller the size of the tile.

In planning the broadcast of the information one has to decide how many informing nodes to accumulate before transmitting the new data. On the one hand accumulating the information saves transmition time. On the other hand getting the information as soon as possible save updating and enables more templets to participate in the propagation process. In our simulation, we adopt the strategy of broadcasting the new information to the neighboring processor whenever a node on the boundary was updated. When the communication overhead is not too large, as in the case of the Meiko transputer board, the number of updates are kept to a minimum since the information delay is very small.

7. Simulation results

Extensive simulations have been carried out on an NS which was tessellated into 145 by 145 nodes. Fig. 6, is a plot of the speedup versus number of processors under different tile sizes. This plot shows that the 4 processors are the natural choice for the two-dimensional NS. For a given patch size, the speedup decreases with the number of processors. This is expected because of the propagated nature of the problem at

hand. The simulation shows, as depicted in Fig. 6, the optimal sizes of the example simulation. These sizes are: 19x19 for 4 processors, 13x13 for 8 processors, and approximatly 9x9 for 16 processors. It shows the general trend that increasing the number of processors decreases the optimal size of the tile.

7. References

- [1] Jones S. T, "Solving Problems involving Variable Terrain. Part 1: A General Algorithm", Byte, Vol. 5, No. 2, February 1980.
- [2] Mitchell J. S. B and Papadimitriou C. H., "The weighted region problem", Tech. Rep., Department op Operations Research, Stanford University, Stanford, CA, 1985.

- [3] Bellman, R., Dynamic Programing Princeton University Press, Princeton, New Jersey, 1957.
- [4] Seymour, P. D., "Disjoint paths in graphs", Discrete Math. 29, pp. 293-309, 1980.
- [5] Wong, Y. F., and Fox, G. C., "Use of Neural Network for Path Planning," Technical Report C3P-784, 1989.
- [6] Fox, G. C., Gurewitz, E., and Wong, Y. F., "A neural network approach to multi-vehicle navigation", SPIE Vol. 1196, pp. 164-169, 1989.
- [7] Salmon J., and Goldsmith J., "A hypercube Ray-Tracer", 3ed Conf. on Hypercube Concurrent Computers an Applications", pp. 1194-1206, Pasadena, CA 1988.

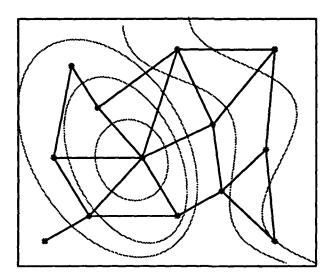
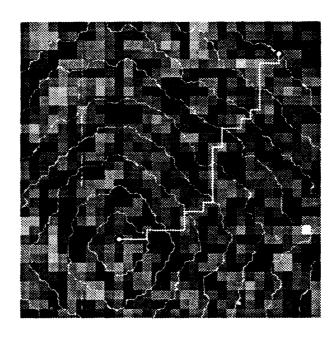


Figure 1. mapping of a terrain onto a graph

^{*} On a leave of absence from NRCN Israel.



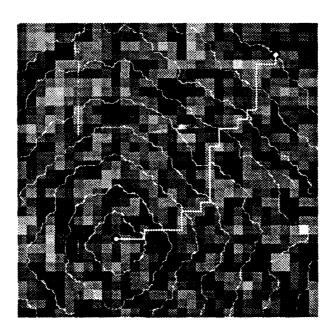


Figure 2. Autonomous vehicle in uncertain environment. The gray level of an area is proportinal to its cost. The white lines are the equi-cost contours After a short travel along the optimal path (a) the vehicle updated its data and determined a new optimal path (b).

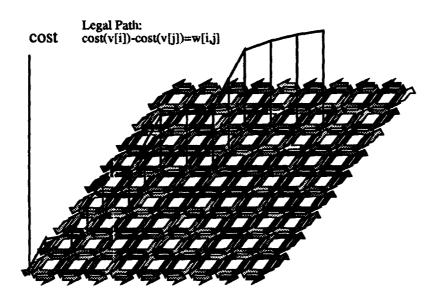


Figure 3. Nodes, terrain's directional values (gray level arrows) and a path in the Cost-Terrain space.

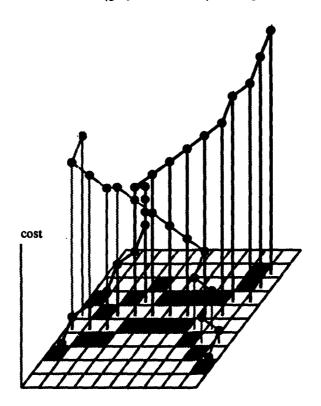


Figure 4. The two-vehicle navigator solution for a conflict imposing terrain and a path in the Cost-Terrain space.

•	1	0	1	0	1	0	1
2	3	2	3	2	3	2	,
•	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
•	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

Figure 5. Scattered decomposition, the basic template of 4 processors is repeated over the terrain.

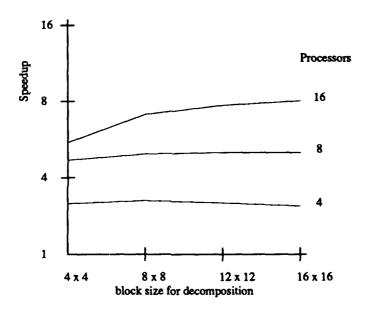


Figure 6. Speedup for decomposition scheme for different block sizes on 16-node Meiko Computing Surface

A Neural Network Approach to Multi-vehicle Navigation

Geoffrey Fox, Eitan Gurewitz*, Yiu-fai Wong

Caltech Concurrent Computation Program California Institute of Technology Mail Code 206-49, Pasadena, CA 91125

Abstract

We develop a neural network formulation for multi-vehicle navigation on a twodimensional surface. here. A time-linking map is generated for each individual vehicle using techniques similar to the known shortest path algorithms for an isolated vehicle. Neural networks are then applied to generate nonconflicting paths minimizing the time of travel.

1. Introduction

This paper presents a neural network approach to the multi-vehicle navigation problem. Here we use the term vehicle to refer to a point which travels on a surface of navigation (NS). Navigation as presented here refers to the determination of a path in the space-cost (time) domain from an origin to a destination point. The surface of the navigation usually has a terrain with position dependent velocities and/or hazards which the vehicle has to consider. The navigator searches for an optimal path on this surface. Optimum here may be with regard to minimal length, minimal time, minimal hazards, etc. Each of these parameters when minimized acts as the cost parameter. To each element of area $dl \times dl$ of the NS is associated the value dt of the cost of traveling the segment length dl on this area. An optimal path between source and destination is the one which yields $min \int_{source}^{destination} dt$.

Navigation problems for one vehicle on a continuous surface as well as on a discrete grid have already been studied and solved^{1,2}. In our paper we consider navigation of more than one vehicle in a two dimensional space, where each vehicle has its own origin and destination. The objective is to navigate the vehicles in a way which minimizes the cost (time) of travel. The time of travel is the time passed between the earliest start time of one of the vehicles to the

last arrival at a destination. When two vehicles or more are involved in the problem, the requirement of collision avoidance may introduce a conflict between the optimal paths of the various vehicles. In order to resolve these conflicts the data base of possible paths is vastly extended and the search for optimal solution is very complicated. In a different study ³ we have directly solved the one- and two-vehicle navigator in a multi speed discrete space. However we did not find a way to extend it as a practical technique for the general multi-vehicle navigator.

A simple NS terrain is defined with a binary speed. On this NS the speed of a vehicle, at each point, is either a positive constant or 0 (for an obstacle). The present study is an attempt to construct a multi-vehicle navigator, in binary speed space, using neural networks. By using neural networks one usually trades an optimal solution accomplished in "infinite" time with "good" solution accomplished in reasonable time.

This study is organized as follows: In section 2 we introduce the cost-surface space and the paths as graphs in this space. The cost-linking map is presented and we discuss the difference between paths solving a one-vehicle navigator and those solving multi-vehicle navigator. In section 3 the neural formulation is presented with the mapping of the space into neural variables, "neural paths" and equations. Section 4 contains the results of our simulations and section 5 the discussion.

2. Paths in the cost-surface space

A descrete representation of the surface of navigation is obtained by mapping the surface onto a graph as follows: a set of points v(x, y) is chosen on the navigation space to be the nodes of the graph. Each node is connected by an edge to every other node which can be reached directly from it. To every edge is assigned a weight

which reflects the cost of traveling it. The edges can have two different weights for traveling it in opposite directions. A path is a sequence of adjacent directed edges from the origin to the destination.

Let us extend the NS terrain to a timesurface space, as shown in Figure 1. A path in this space is a sequence of edges, monotonic in t, between the source and destination. However, a legal path is one which obeys the restrictions set by the terrain. In order to get only legal paths, we construct a time-linking map³. This map assigns to every node the minimal time of travel needed to reach it from the origin. Using this map one can construct a graph of all the optimal paths from the source to all the nodes. This map specifies the t-coordinate of each node of the graph in the time-NS space. An optimal path for one vehicle, in the time-surface space, is single-valued in v(x, y) and t. Namely, there is a one to one correspondence between v(x,y)on the path and t. When more than one vehicle are involved each one of them has its own linking map. However, the optimal paths of two different vehicles may conflict. To avoid such a conflict one of the vehicles may be requested to postpone its arrival to or to detour the point of conflict. This imposition introduces paths which are not single valued in v(x, y) and t as illustrated in Figure 2.

3. Neural formulation

Neural networks have been studied as an approach to various hard (NP-complete) optimization problems. Various applications have been investigated and explored^{4,5} since the work of Hopfield and Tank⁶. Here, we explore the possibility of using these massively parallel networks for the multi vehicle navigation problem.

The paths of the vehicles, as discussed above, are viewed as trajectories in the space-time. The space-time is mapped into neural variables in the following way: it is divided into a regular three dimensional lattice (x, y, t). (For notational simplicity, we denote (x, y) by the vector x subsequently.) To each unit cell we associate a neural variable $\eta_i(x,t)$ whose desired

value, at stable state, is:

$$\eta_i(x,t) =
\begin{cases}
1, & \text{if vehicle } i \text{ is at position } x \\ & \text{at time } t; \\ 0, & \text{otherwise.}
\end{cases}$$

A path is a sequence of neural variables with $\eta_i(x,t)=1$ where t ranges from 0 to T and T is the time this path is traveled. A neural network is set up such that the neurons converge to a stable state which determines the paths as illustrated in Figure 3. A common practice in optimization by neural networks is to choose an energy function. However, finding the shortest paths is an iterative process, which makes our energy function time-dependent. Therefore, instead of minimizing an energy function we directly write down the equations relating the input "voltage" of the neurons to their output voltage. These equations impose the desired behavior of the neurons. Specifically, we have

$$du_i(x,t)/d\tau = C_1 + C_2 + C_3 + C_4 + C_5$$

where the first term evaluates the propagation of the path from the present position in the forward direction. The second term evaluates it with respect to the backward direction. The third term avoids head-on collision and the fourth term avoids swapping which occurs when two vehicles adjacent to each other switch positions. The fifth term forces one of the neighbors of an "on" neuron to be on, i.e. enforces continuity of the paths. In terms of neural variables, the dynamical equations is as follows:

$$du_{i}(x,t)/d\tau = gate(i)\Big($$

$$-u_{i}(x,t)$$

$$+A_{1} \sum_{y \in Nb(x)} \eta_{i}(y,t-1)W_{xy}$$

$$+A_{2} \sum_{y \in Nb(x)} \eta_{i}(y,t+1)W_{y,x}past_{i}(x,t)$$

$$+A_{3} \sum_{k} \eta_{k}(x,t)g(s_{i}(t)-s_{k}(t))+$$

$$\begin{split} & + A_4 \sum_k \sum_y \eta_k(x,t-1) \eta_i(y,t-1) \eta_k(y,t) \cdot \\ & \prod_{x' \in Nb(x), x' \neq y} \left(1 - \eta_i(x',t-1) (W_b - W_{xx'})/2 \right) \cdot \\ & \prod_{x' \in Nb(y), x' \neq x} \left(1 - \eta_k(x',t-1) (W_b - W_{yx'})/2 \right) \\ & + A_5 \sum_{y \in Nb(x)} f(\eta_i(y,t-1)) \cdot \\ & \prod_{x' \in Nb(y)} (1 - f1(\eta_i(x',t))) \right) \end{split}$$

Where every term $A_i\{\cdots\}$ corresponds to the respective term C_i . $\eta_i(x,t) = h(u_i(x,t)), h(\cdot)$ is a sigmoid function giving the relation between the input and the output voltage of a neuron. Nb(x) = neighborhood of x. W_{xy} is the cost of travel from x to y with regard to the destination of the vehicle. Namely, $W_{xy} = T(x) - T(y)$ where T(u) is the time-linking map value of the node u. If T(y) < T(x), it encourages the forward (in time) propagation of the path from x to y. $past_i(x,t) = \sum_{y \in Nb(x)} \eta_i(y,t-1)$ gates the backward propagation. A neuron is affected by the future information only if it is a continuation of a path. $s_i(t) = \sum_x \eta_i(x,t)$, and g(.)is another sigmoid function which says that in case of collision, the vehicles with more possible paths should give way. The swapping term is most complicated. We leave out the detailed explanation except saying that $f(\cdot)$ and $f(\cdot)$ are appropriately chosen highly nonlinear functions. Lastly, $gate(i) = \prod_{\tau \leq t} \eta_i(x_{di}, \tau)$ which stops the signal propagation for vehicle i once its destination x_{di} has been reached.

In the equations above, we encourage all possible paths to be stored in the states of the neurons. The redundancy in the formulation makes this possible. When the destinations are reached, we backtrack and choose one of the best paths computed by the network.

It is obvious that in the absence of collision, the paths obtained are the original optimal paths for a single vehicle where collisions are not considered.

Since the problem is inherently time dependent, the neuronal states at large t naturally wait for the information from neurons at smaller t. We may as well solve the equation for a fixed time window ω , namely we compute the paths

for the next ω moves. Then we repeat the procedure, calculating the paths piecewise until the destinations are reached.

4. Simulation results

We numerically integrated the above dynamical system, using a simple Euler method, in which case, synchronization does not have to be exactly enforced. Recall that an Euler solver for a differential equation dx/dt = f(x) is an iterative mapping: $x_{i+1} = x_i + \epsilon f(x_i)$. This, together with the locality of the computational stencil enables us to parallelize the above algorithm very efficiently. If we go back to the equations above, the only global computation is computing $s_i(t)$, which can be obtained by locally updating the sum within each processor and combining the result in a binary tree. By iteratively solving a differential equation, exact synchronization is not needed because the dynamics is continuous. In a similar study¹¹, but slightly modified dynamic equations, almost a perfect speedup was obtained when it is implemented on the Meiko Computing Surface, a parallel machine with up to 32 transputer nodes as illustrated in Figure 4. The differential form also introduces some cooperation into the algorithm. This can be observed in the conflicting regions, like head-on collision and swapping, in which case the neurons iteratively adjust their values, trying to resolve the conflict.

5. Discussion

The neural net yields paths which slightly deviate from the optimal one-vehicle paths. This is because it is dominated by two nain forces: one is the collision avoidance force and the other is the single vehicle optimal paths attractors. These attractors are the graphs determined by the linking map from a node to the destination. In our study of the two vehicle analytic navigator³ we are using an algorithm which updates this map. Applying this idea to the neural net system can improve the solutions obtained above. The neural net four vehicle navigator performs well, see Figure 3. However, with more vehicles and various possible paths for each it may perform less satisfactorily. Clearly we only presented a very initial study here. We need to look at much more complex problems including three dimensional navigation. We are

also looking into the elastic net ideas of Durbin and Willshaw ⁷ as interpreted by Simic⁸ into neural networks. There are important analogies between track finding ⁹, computer vision and navigation which we are exploring in an integrated research program¹⁰.

6. Acknowledgements

The work reported here is funded by the Department of Energy under grant DE-FG03-85ER25009, Program Manager of the Joint Tactical Fusion Program Office and National Science Foundation under grant EET-8700064.

7. References

- M. Sharir & A. Schorr "On Shortest Paths in Polyhedral Spaces", SIAM J. Computing, 15,1, pp. 193-215, 1986.
- 2. J. S. B. Mitchell and C. H. Papadimitriou, "The Weighted Region Problem", Technical report, Dept. of Operations Research, Stanford University, 1986.
- 3. E. Gurewitz G. Fox, and Y. F. Wong "Parallel Algorithms for One and Two-vehicle Navigation" in this proceedings.

- 4. D. Z. Anderson, "Neural Information Processing Systems," American Institute of Physics, New York, 1988.
- "IJCNN Proceedings," Washington D.C., 1989.
- 6. J. J. Hopfield & D. W. Tank, "Biol. Cybern. 52, pp. 141-152, 1985.
- R. Durbin and D. Willshow, Nature, 326, 689-691, 1987.
- 8. P. Simic, "Statistical Mechanics as the Underlying Theory of 'Elastic' and 'Neural Optimizers'," Technical Report C3P-787, contribution to 1989 SCS Eastern Conference, Florida, 1989.
- 9. Geoffrey Fox, "A Note on Neural Networks for Track finding," Technical Report C3P-748, 1989.
- G. C. Fox, W. Furmanski, A. Ho, J. Koller, P. Simic, Y. Wong, "Neural Network and Dynamic Complex Systems," Technical Report C3P-695, 1989.
- Y. F. Wong and G. C. Fox, "Use of Neural Network for Path Planning," Technical Report C3P-784, 1989.

^{*} On a leave of absence from the Physics Department NRCN Beer-Sheva Israel.

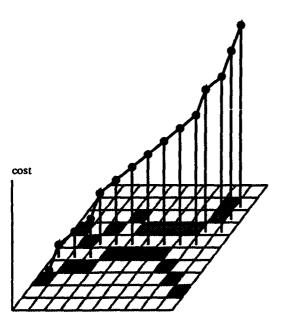


Figure 1. A path in the cost-terrain space

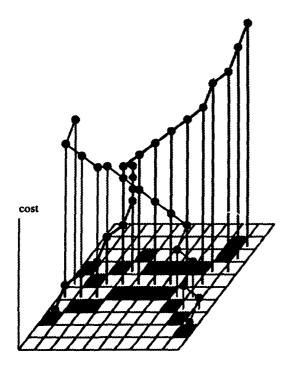


Figure 2. The two-vehicle navigator solution for a conflict imposing terrain and a path in the Cost-Terrain space.

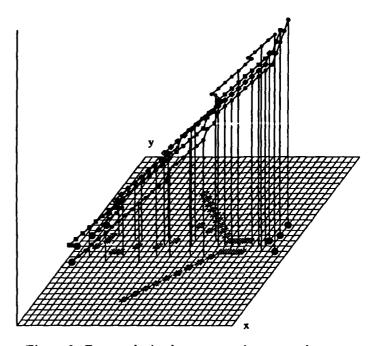


Figure 3. Four paths in the cost-terrain space calculated by the neural net

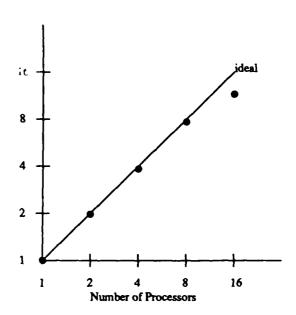
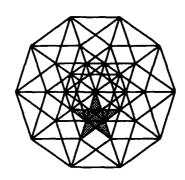


Figure 4. Speedup for 4 vehicle navigator running on 16-node Meiko Computing Surface



The Fifth Distributed Memory Computing Conference

6: Data and Image Processing

A Connectionist Technique for Data Smoothing

Ron Daniel Jr. and Keith Teague
Oklahoma State University
School of Electrical and Computer Engineering
202 Engineering South
Stillwater, OK 74078

Abstract

Filtering data to remove noise is an important operation in image processing. While linear filters are common, they have serious drawbacks since they cannot discriminate between large and small discontinuities. This is especially serious since large discontinuities are frequently important edges in the scene. However, if the smoothing action is reduced to preserve the large discontinuities, very little noise will be removed from the data.

This paper discusses the parallel implementation of a connectionist network that attempts to smooth data without blurring edges. The network operates by iteratively minimizing a non-linear error measure which explicitly models image edges. We discuss the origin of the network and its simulation on an iPSC/2. We also discuss its performance versus the number of nodes, the SNR of the data, and compare its performance with a linear Gaussian filter and a median filter.

Introduction

A common operation in image processing is filtering to remove noise. One of the simplest methods is to implement a linear low-pass filter by convolution with a Gaussian, or other, kernel. The availability of dedicated convolution processors makes this option especially attractive for many machine vision systems. Unfortunately, the linear filter has serious drawbacks. It cannot discriminate between large discontinuities and small discontinuities. Nor can it model the structure of the data to discriminate between correlated discontinuities, such as edges, and random noise. Large correlated discontinuities are frequently edges of objects in the scene, which convey considerable information. Reducing the amount of smoothing in order to preserve important discontinuities also reduces the amount of noise removed. Thus, linear filtering is a compromise between preserving large discontinuities while still removing noise from the data. A good compromise can be very difficult to strike.

As general-purpose parallel processors become more widely available, and as their cost continues to decline, the performance advantage of convolution hardware will be reduced. This will allow more sophisticated filters to be used without an unacceptable performance penalty. This paper discusses the parallel implementation of a 'neural network' approach to the data smoothing problem. The smoothing technique is

based on iterative minimization of a non-linear error measure. The error measure has several components. Squared error of the solution from the input data and smoothness of the solution are two of the components. These are very common [1,2]. The unusual portion of the error measure is the introduction of 'breakpoints' across which the smoothing terms have no weight. This modification of the surface reconstruction problem appears to have first been used in [3]. These terms model edges in the image and allow us to smooth noisy data without blurring the edges of objects in the image. A one-dimensional version of the resulting energy measure was presented in [4] as:

$$E(f, h) = \sum_{i} (f_{i+1} - f_{i})^{2} (1 - h_{i})$$

$$+ C_{D} \sum_{i} (f_{i} - d_{i})^{2} + C_{L} \sum_{i} h_{i}$$
(1)

where f_i is the smoothed output value from size i, h_i is a binary variable indicating the presence or absence of a discontinuity between units i and i+1, d_i is the input data to unit i, C_D is the cost of getting away from the data relative to the unit weight of the interpolation term, and C_L is the cost of inserting a discontinuity. To allow for sparse data, the summation is only taken over those points where $d_i \neq 0$. C_D depends upon the signal to noise ratio of the input data.

The interpretation of this equation is that if $(f_i - f_{i+1})^2 > C_L$, then it is cheaper to pay the price of inserting the discontinuity than continuing to smooth over the large disparity in function values. The data filtering is performed by adjusting the f_i and h_i to minimize E. Since the discontinuity terms introduce local minima into the cost surface, standard minimization algorithms will not work very well. Simulated annealing was used in [3] to perform the minimization.

Koch Network

Koch, et. al. [4], present another method for minimizing (1) based on the work of Hopfield [5,6]. Hopfield suggested solving optimization problems by changing the binary variable, h_i, to a continuous [0,1] variable that is a nonlinear function of an underlying state variable. Additional terms are introduced into the energy function to force the solution toward 0 or 1. The

minimization is then carried out by having a network of units, one for each of the f_i and the h_i , which update their value by the rules:

$$\frac{df_i}{dt} = -\frac{\partial E}{\partial f_i}$$
, and $\frac{dm_i}{dt} = -\frac{\partial E}{\partial h_i}$ (2)

where m_i is the state variable underlying h_i . We are not showing the time constants that set the rate of change of the units. The nonlinear function, g(), is typically the sigmoid nonlinearity:

$$h_i = g(m_i) = \frac{1}{1 + e^{-2\lambda m_i}}$$
 (3)

Because of the update rule, each site takes a small step down the gradient of the cost function. While each site must take many steps to reach the minimum of the function, the steps can proceed in parallel. Therefore, for a large number of sites, the total time to perform the minimization should be reduced.

The function used in this study was proposed in [4]. It is for filtering two-dimensional data, as opposed to the one-dimensional data smoothing that would be performed by (1). The presence of a 'horizontal' break, one between $f_{i,j}$ and $f_{i,j+1}$, is indicated by h_i . Vertical breaks, between $f_{i,j}$ and $f_{i+1,j}$, are indicated by v_i . The function used is:

$$E = E_I + E_D + E_L + E_G, \qquad (4a)$$

$$E_{I} = \sum_{i,j} (f_{i,j+1} - f_{i,j})^2 (1 - h_{i,j}),$$
 (4b)

$$E_D = C_D \sum_{ij} (f_{i,j} - d_{i,j})^2$$
 (4c)

$$E_{L} = C_{V} \sum_{ij} h_{ij} (1-h_{i,j})$$
 (4d)

+
$$CP \sum_{ij} h_{ij} h_{ij+1} + CC \sum_{ij} h_{ij}$$

+
$$C_{L_{ij}}^{\sum} h_{ij} [(1-h_{i+1,j}-v_{i,j}-v_{i,j+1})^2]$$

+
$$(1-h_{i-1}j - v_{i-1}j - v_{i-1}j+1)^2$$

$$E_G = C_G \sum_{ij} \int_{0}^{h_{ij}} g_{ij}^{-1}(h_{ij}) dh_{ij}$$
 (4e)

where fij is the interpolated surface and hij and vij are the horizontal and vertical line processes. Note that the energy expression above is only for hij. The other half of the expression can be obtained by replacing hij with vji, substituting i for j and vice versa. The first term in EL forces hii to either 0 or 1, the second term penalizes the formation of parallel lines, the third term is the constant price that is paid for introducing discontinuities, and the fourth term is an interaction term which favors continuous lines while penalizing multiple line intersections, line crossings, or discontinuous line segments. The line outputs hij and vii are functions of internal state variables. Since the discontinuity terms are asymtotic to 0 and 1, the update rules would drive these state variables to $\pm \infty$ in a futile attempt to drive the visible outputs to 0 or 1. The EG term prevents this by penalizing excessive values for the state variables. The smoothed data output and the internal state variables are updated according to:

$$\frac{df_{i}}{dt} = -\frac{\partial E}{\partial f_{i}}, \quad \frac{dm_{i}}{dt} = -\frac{\partial E}{\partial h_{i}}, \quad \frac{dn_{i}}{dt} = -\frac{\partial E}{\partial v_{i}} \quad (5)$$

Parallel Implementation

The network was simulated on an iPSC/2 SX (Weitek FPUs) under HIP, the Hypercube Image Processor [7]. HIP is system for interactive image processing, as well as a framework for developing parallel image processing algorithms. By providing predefined image decompositions, I/O procedures, and a body of image processing functions, HIP reduces the effort required to develop parallel image processing algorithms. HIP supports floating-point image buffers in addition to character and integer types. It also supports multi-spectral image buffers. HIP's image buffers have a simple decomposition. Each image is divided into as many horizontal strips as there are nodes and each strip is given to a different node. Each strip is provided with a border of data to hold initial conditions for convolutions and similar neighborhood operations. The top and bottom of the border is updated from the active region of the two neighboring nodes. The sides of the border are updated by replicating the first and last

The network was simulated by using a floating-point image buffer with 5 spectral bands. The first band holds the state of the smoothed output values, the second and third hold the horizontal and vertical discontinuities, while the fourth and fifth hold the state variables underlying the horizontal and vertical discontinuities, m_i and n_i. The input data, d_i, comes from a seperate image buffer. At each iteration, all the sites in the smoothed output layer are updated according to the update rules in (5). At the end of each iteration, the border data is updated with the new values from the

neighboring nodes. The discontinuities are not updated every iteration. The frequency of their update is controlled by a command-line option. For the results presented in this paper, they were updated every 4 iterations.

The depth layer is initialized with the values of the image to be smoothed, while the edges and their underlying state variables are intialized to the middle of their range. The minimization procedure was terminated when the value of the cost function stopped decreasing. The number of iterations this took depends upon the data and the settings of the time constants not shown in equation (2), but was generally between 10 and 20 iterations.

Performance

There are several facets of the network's performance to characterize. We are, of course, interested in how well it parallelizes. We must also be interested in the quality of filtering it performs and its ease of use. We will discuss these in order.

Since the filter is iterative, it is impossible to predict in advance how many iterations will be required for termination. For this reason we will report the execution time in two fashions. When we look at how well the network parallelizes, we report the time taken to complete a single iteration that updates the depth and discontinuity layers. When we compare the filter with other filters, we will report total times.

Since each site is updated only as a function of the sites in a four nearest neighborhood, we would expect to see nearly linear speedup as we increase the number of nodes. We would also expect the execution time to be directly proportional to the size of the data set. This is just what is shown in figures 1 and 2, which report the time to complete one iteration as a function of the number of nodes and the size of the image. Figure 2 uses a log scale, and shows almost perfectly linear behavior. The speedup coefficients are 0.86, 0.91, 0.94, and 0.97 for the 64 .. 512 image sizes, respectively.

Note that the data points missing from figures 1 and 2 are due to insufficient node memory to hold large images on few nodes. The log scale shows an anomaly for the 64x64 image on 32 nodes, which requires some explanation. Recall the border of initial conditions data that HIP provides in its image decomposition. If the border has more lines in it than the neighboring node has in its active region, several communication steps will be needed to update the borders. HIP determines if this is the case and uses a fast border update if possible, and a slow-but-sure update if not. A 64x64 image on 32 nodes has only two rows in its active region, so we are seeing a different border update procedure. The alert reader may be asking why more than a single row in the border is needed. Actually, it is not. This is a just-discovered coding error in HIP which will be corrected before it is released to the iSC User Group library.

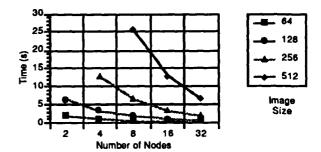


Figure 1: Time / Iteration vs. Number of Nodes and Image Size

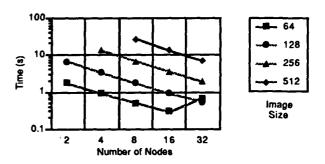


Figure 2: Time / Iteration vs. Number of Nodes and Image Size (Log Scale)

The timings above were all for a single iteration of the network. We also need to know the total time for the network to converge. These are given below in table 1 for the 16 node case.

The Koch network is not the only non-linear data filter available. We decided to compare its performance with two other filters, a 5x5 linear Gaussian low-pass filter and a 5x5 median filter. An artificial image was generated and corrupted with different amounts of noise. The three filters were applied and their execution time noted. Finally, the sum of the squared errors were computed. The image used is shown below in figure 4a. It is 128x128 with the darkest gray level at 25 and the brightest at 229, with the other 3 levels at 76, 127, and 178. The noise added was uniformly distributed and 0mean. Two magnitudes were used, from -25..25 and -12.5..12.5. These correspond to SNRs of 13 dB and 20 dB, respectively. The outputs of the filters for the 13 dB SNR are shown in figures 4b..4e. Figure 4f shows the horizontal discontinuities detected by the Koch network. the vertical discontinuities are similar. The filters were also applied to the uncorrupted image to see what damage they would inflict upon perfect data. The sums of the squared errors (SSE) are plotted below in figure 3. A perfect filter would be a flat line at the bottom of the graph. The steepest line is the SSE for the unfiltered, noisy, image. Data points above this line show a filter that is doing more harm than good.

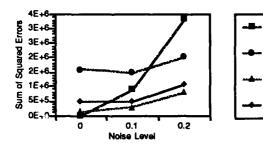


Figure 3: Sum of Squared Error vs. Filter Type and Noise Level

This figure shows that the Koch network performs much better than the linear Gaussian filter, but not as well as the median filter. It also takes much longer to execute, as shown below in table 1. These are the times on 16 nodes for the 13 dB SNR images.

Table 1: Execution Times of Filters (sec.)

Image Size	Gaussian	Median	Koch 10.47 19.48	
64	.577	.483		
128	1.1	.883		
256	2.26	4.14	41.91	
512	3.54	14.18	63.86	

Conclusions

Neural network approaches to machine vision tasks are the focus of a great deal of research interest. Part of the reason for this is because of their massive parallelism. Their fine-grained structure allows them to be mapped onto almost any parallel architecture, although networks that are almost completely interconnected will pay a performance penalty. Those networks with restricted interconnections between units, such as the Koch network, are especially easy to implement on distributed-memory computers. This is shown by the excellent speedup as the number of nodes increased and the O(N) behavior as the data size increased.

While the Koch network is easily parallelized, so are many standard filters used in image processing. The median filter performs better than the Koch network by all the measures we used. It also has the advantage of a predictable execution time. Since the Koch network is iterative, one can never be entirely sure how long it will take to complete. The Koch network does have an advantage over the median filter for the case where a dedicated VLSI implementation is considered. The update rules in (5) can be implemented by analog computation, which would offer a tremendous performance improvement over the digital multiply and add.

Another disadvantage to the Koch network is that it is hard to use. There are many parameters that must be balanced to achieve good performance and no a priori

method for determining them. Furthermore, the parameters are sensitive to the magnitude of the data. 0..1 data requires different parameters than 0..255.

Figure 4f shows another problem of the network. Its small neighborhood size makes it sensitive to tiny regions of correlated noise. This can be overcome by the use of multiresolution techniques [8], which seem to give excellent results. We hope to add image pyramid buffers to HIP in the future in order to attempt to duplicate the results in [8].

References

- [1] Poggio, T., Torre, V., Koch, C. (1985) Computational Vision and Regularization Theory, Nature, Vol. 317: 314-319, 26 Sept. 1985.
- [2] Narayanan, K.A., O'Leary, D.P., Rosenfeld, A. (1982) Image Smoothing and Segmentation by Cost Minimization, *IEEE Trans. Sys.*, Man, and Cyber., SMC-12(1): 91-96, January.
- [3] Geman, S., Geman, D. (1984) Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images, *IEEE Trans. Patt. Anal. and Mach. Intell.*, PAMI-6(6): 721-741, November.
- [4] Koch, C., Marroquin, J., Yuille, A. (1986) Analog 'Neuronal' Networks in Early Vision, *Proc. Natl. Acad. Sci. USA*, Vol. 83: 4263-4267, June.
- [5] Hopfield, J.J. (1984) Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons, Proc. Natl. Acad. Sci. USA, Vol. 81: 3088-3092, May.
- [6] Hopfield, J.J., Tank, D.W. (1986) Computing with Neural Circuits: A Model, Science, Vol. 233: 625-6338, August.
- [7] Daniel, R. Jr., Carter, M.B., and Teague, K.T. (1989) Design and Implementation of a Parallel Image Processing System for the iPSC/2. In Proceedings of the Fourth Conf. on Hypercubes, Concurrent Computers, and Applications (HCCA4), Monterey, CA, April.
- [8] Battiti, R. (1999) Surface Reconstruction and Discontinuity Detection: A Fast Hierarchical Approach on a Two-Dimensional Mesh. In Proceedings of the Fourth Conf. on Hypercubes, Concurrent Computers, and Applications (HCCA4), Monterey, CA, April.

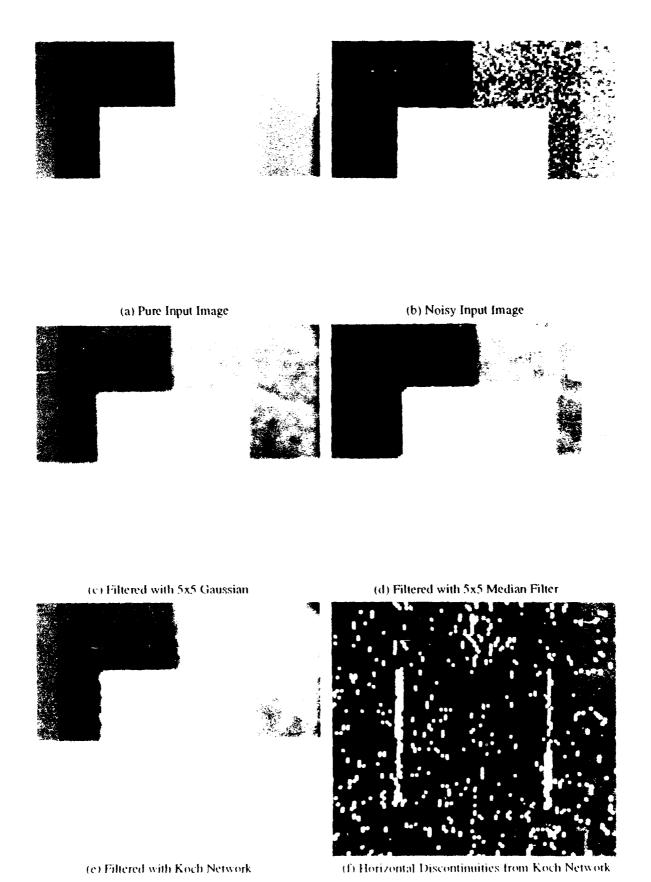


Figure 4: Input Images and Filtered Outputs

Component Labeling Algorithms on an Intel iPSC/2 Hypercube *

Babak Falsafi

Russ Miller

Graduate Group in Advanced Scientific Computing
226 Bell Hall
State University of New York
Buffalo, NY 14260
miller@cs.buffalo.edu

ABSTRACT

One of the intermediary stages of image analysis in vision is the process of component labeling. Given a digital black/white image distributed throughout the nodes of an Intel iPSC/2 hypercube, the objective of this research is to develop and implement efficient parallel algorithms for labeling the (black) connected components. The basic solution strategy is based on divide-and-conquer, in which each node initially labels the subimage that it is responsible for. The results of the local labeling are then combined using boundary-overlapping resolution strategy. In an

effort to develop algorithms that are both time and space efficient, we consider manipulating various data structures, using a variety of sequential and parallel component labeling schemes, and performing load balancing techniques to maximize parallelism. The images currently under experiment range from real pictures extracted from scanners, to medical X-rays, to digital images generated with respect to certain constraints. Experimental results, analysis, and interpretation of various algorithms are presented.

^{*} This work was partially supported by NSF grants IRI-8800514 and ASC-8705104.

1. Introduction

A digitized black/white image, also known as a binary image, consists of a two-dimensional array of pixels with foreground pixels having the value 1 (black), and background pixels having the value 0 (white). A (connected) component in an image is defined to be a set of maximally connected foreground pixels, where two pixels are connected if and only if they are adjacent. There are two common definitions of adjacency. In the first definition, known as 4-connectivity, two black pixels are defined to be adjacent if and only if one pixel is directly above, below, to the left or to the right of the other pixel. In the second definition, known as 8-connectivity, two black pixels are defined to be adjacent if and only if one is one of the eight closest pixels of the other. The problem of component labeling is to assign a unique label to every component in an image so that every pixel is assigned its component's label.

Our goal is to design algorithms for labeling the components of a digitized image on a hypercube. The algorithms presented assume that the binary image to be processed has been partitioned into vertical slices and distributed throughout the nodes so that each node is responsible for a unique strip as illustrated in figure 1.

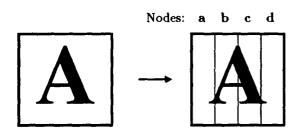


Figure 1. Dividing the image and distributing it amongst the nodes

We perform component labeling based on divideand-conquer in two major steps; the first step consists of a sequential component labeling algorithm that is applied to each vertical slice (base case), the second step consists of a strategy for resolving the conflicts between the boundary labels of the neighboring subimages (conquer).

In section 2, we give a brief description of an approach to the sequential step. Section 3 covers two different algorithms for solving the problem of

boundary-overlap resolution. Section 4 discuses an algorithm for the path resetting problem in a resolution table based on a series of union/find operations. Load balancing techniques are presented in section 5, and finally the timing results are given at the end of the paper.

2. Sequential Algorithm

The sequential labeling algorithm that we use is based on a two-pass labeling scheme similar to [1]. During the first pass the image is examined row by row (top-down, left to right) while labeling the foreground pixels as follows.

- Assign a new label to a foreground pixel that is not connected to any other previously labeled foreground pixel.
- ii) Assign the same label to a foreground pixel that is connected to a previously labeled foreground pixel.
- iii) If there are two adjacent pixels with different labels, create an entry in the resolution table indicating that the two labels must be resolved during the second pass.

Once the first pass is complete, some of the components are not labeled in a consistent fashion (as discussed in iii above). The reset_paths procedure, discussed in section 4 takes the resolution table and resolves the conflict among the labels producing a final table which contains a list of the labels resolved and their new values.

During the second pass through the image, as each foreground pixel is examined a search is performed in the resolution table to see if the label of the pixel should be updated.

3. Conflict Resolution

Since the sequential algorithm is applied to each subimage independently, it is expected that inconsistencies in the assigned labels exist for objects that lie across the boundaries of subimages.

In this section we describe algorithms to resolve the inconsistencies at the boundaries of subimages. We first turn our attention to a simple approach to solve the problem, and later present load balancing techniques to speed up the process.

Conceptually we arrange the nodes of the hyper-

 In doing step ii, it is necessary to have a list of pixel positions corresponding to a label in a given array of boundary pixels.

The table is therefore initially set up to contain an entry corresponding to each label and sorted by increasing value of the labels so that adding entries would consist of performing a search on the label and assigning a new value to it. For every array of boundary pixel there is a list of labels created with each label having a list of pixel positions assigned that label.

The above algorithm can then be implemented as procedure $resolve_overlaps$. Given a foreground pixel at position x labeled l, two arrays of boundary pixels col1 and col2, two lists of labels list1 and list2, and a resolution table T the procedure finds the pixels adjacent to it across the boundary and resets their label in the table to l (we refer to the adjacent pixels across the arrays as "neighbors").

resolve_overlaps

```
procedure resolve_overlaps(col1,col2,
                         list1, list2, T, l, x)
  for each i in neighborsof(x, col2) do
     k := label of i
     tempk := k
     if (k, newk) \in T
       k := newk
     if(k \neq l)
       add (k,l) to T
       setofpixels := search(list2, tempk)
       for each j \in setofpixels do
          resolve_overlaps(col2,col1,
                    list2, list1, T, l, j)
       end for
     end if
  end for
```

A complete resolution table can then be formed by successive calls to resolve_overlaps for all the foreground pixels in one of the arrays as follows:

```
for each black pixel x in col1 do l := label of x if (l, newl) \in T l := newl resolve_overlaps(col1, col2, list1, list2, T, l, x) end for
```

3.1.2 Alternative Approach

The alternative approach proceeds by pairing labels of adjacent foreground pixels in the two arrays of boundary pixels and forming a table (figure 3). Next, the table is sent to the path resetting algorithm discussed in section 4, whereby a resolution table is obtained. The advantage of this over the initial approach is that the find and union operations used in the path resetting algorithm use path compression so that successive search operations in the forest can be performed more efficiently.

A variation of this algorithm that we have implemented avoids making some duplicate entries in the table as much as possible so that the input to the path resetting algorithm would be smaller. This is done by comparing the current label pair with the most recent entry inserted in the table, and insert the current entry only if it is different.

In figure 3 an example of an input to the path resetting algorithm set up by above scheme is shown. The two boundary pixel arrays of figure 2b are considered, and the duplicate entries are omitted.

11	12
5	2
5	3
6	4
6	2

Figure 3. Table set up for the path resetting algorithm

The conflict resolution table given in figure 2c is the result of the path resetting algorithm applied on figure 3.

4. Path Resetting

The sequential algorithm discussed in section 2 and the boundary resolution algorithm discussed in section 3.1.2 produce tables of the form illustrated in figure 3. In these tables for every label pair (l1, l2) there can be a corresponding label pair (l1, l3) or (l2, l3). In either case the labels l1, l2 and l3 are to be resolved to a unique label by modifying the table to contain the label pairs (l1, l1), (l2, l1) and (l3, l1) instead. Path resetting is the process of grouping the labels into disjoint sets where an element belongs to a set if and only if there is an entry corresponding to that label and another member of the same set in the

cube in a linear array, with each node having a labeled slice of the image. The algorithm proceeds by entering a loop. In each iteration, the nodes in the linear array are paired, the boundary-overlap resolution algorithm is applied to each pair, the boundaries are consequently updated, and then pairs of slices are combined into bigger slices, thereby reducing the number of slices to be processed by 2. Figure 2a illustrates the array of processors for a 4-node hypercube. In the first step all of the 4 nodes are participating in the process of conflict resolution, and then groups of two are formed in the second step, where conflict resolution only occurs in the nodes lying at the boundaries of the bigger slices (i.e. b and c).

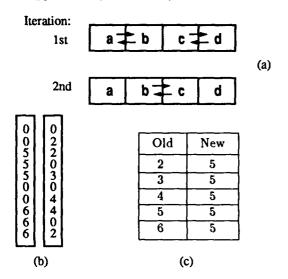


Figure 2. Conflict resolution occurring in each iteration (a) linear array of nodes (b) boundary pixel arrays (c) resolution table

3.1 Boundary Resolution Algorithm

In each iteration of the conflict resolution algorithm labels are resolved at the boundaries of each pair of slices in the linear array of nodes as discussed previously. As the boundary resolution occurs between two of the nodes located at the boundaries of two vertical strips, one of the nodes sends a boundary pixel array to the other node, which is in turn responsible for performing the boundary-overlap resolution and distributing the results to all the nodes in the two vertical strips. For example, consider the configuration of the linear array of nodes in the second iteration illustrated in figure 2a. Node b sends the rightmost array of boundary pixels to node c. Node c

runs the boundary-overlap resolution algorithm and passes the results to nodes a, b, and d.

The boundary-overlap resolution algorithm takes as input two arrays of boundary pixel labels and returns a resolution table for updating the boundary pixels. An example of arrays of boundary pixel labels used in the process of conflict resolution is illustrated in figure 2b, where a 0 denotes a background pixel. The resulting resolution table for this example is shown in figure 2c.

We present two different approaches for solving the problem of boundary-overlap resolution.

3.1.1 Initial Approach

Initially, we used a recursive algorithm for resolving the inconsistencies in the labels assigned to adjacent foreground pixels across the boundary (note that adjacency is defined by δ -connectivity). The idea is to take foreground pixels from one array and a table T (initially empty) and perform the following for each pixel x:

- i) Let l be the label of x. If there is a (l, newl) in T let l := newl.
- ii) For each y an adjacent pixel of x:
 - a) Let k be the label of y. If there is a (k, newk) in T let k := newk.
 - b) If $k \neq l$, add (k,l) to T, and for all foreground pixels having the same label as y repeat step ii with x = y.

The labels of the foreground pixels change as the algorithm proceeds. Therefore whenever labels of foreground pixels are to be used, there is a search performed on the label in the resolution table to check if there is a new value associated with it.

Applying the above strategy to the arrays of figure 2b, we will have l=5 for the third foreground pixel from the top in the left array. The adjacent pixels with inconsistent label are both labeled 2. The set of indices of all the pixels labeled 2 is $\{2,3,10\}$. For each of the elements of the set step ii is repeated with l=5.

Formulating the above into a procedure, we note that:

 If labels are to be searched, the table must be kept sorted by increasing order of l1 for each (l1, l2) entry so that binary search could be applied. table.

Given a resolution table T, procedure reset_paths proceeds by sorting the labels in T ascending order and placing them in a forest F where each label is placed at the root of a tree. Grouping of labels into disjoint sets can then be achieved by a series of find and union operations [2]. The end result is then obtained by taking the root of each tree in the forest as the representative of the tree and create a new table by pairing every label in each of the trees with its root.

reset_paths

```
init_forest(F,T)
for each (l1,l2) ∈ T
    root1 := find(F,l1)
    root2 := find(F,l2)
    if root1 ≠ root2 then
        union(F,root1,root2)
end for

tempT := an empty table
for each tree S in F do
    root := rootof(F,S)
    for each l∈ S do
        add (l,root) to tempT
    end for
return tempT
```

5. Load Balancing

In order to balance the load on all the processors, the process of boundary resolution between two vertical strips performed by a single processor discussed in section 3.1 can be divided among all the nodes within each vertical strip (figure 2a). By allowing every node in a group of nodes in a vertical strip to perform boundary resolution on a subdivision of the two boundary pixel arrays, intermediary resolution tables can be formed which are then merged together to result a final resolution table. The finale resolution table is then distributed among the nodes in the group.

As an example consider the configuration shown in figure 2a, where boundary resolution is occuring between nodes b and c in the second iteration of the algorithm. We divide the two arrays of boundary pixels by the number of nodes and distribute them as in figure 4. The subparts of the boundary arrays have

a pixel-long overlapping region, (i.e. the last entry in the arrays sent to one node is the same as the first entry of the arrays in the neighboring node) so that the diagonally adjacent pixels at the end of each subarray be considered in the process of boundary resolution.

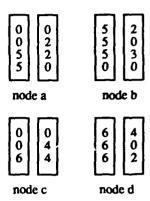


Figure 4. Dividing the boundary pixel arrays into subparts

6. Timing Results

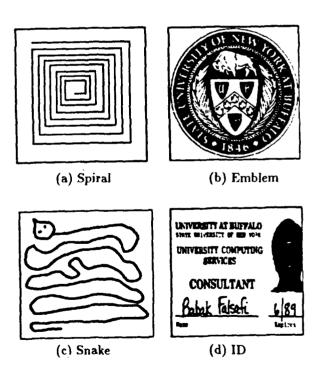


Figure 5. The images for which timing results were measured.

Figure 5 illustrates the images, for which the timing results are given in this section. The images are 512x512 matrices of black and white pixels. The algorithms are exmained on an Intel iPSC/2 Hypercube.

The runtime of the sequential algorithm is proportional to the number of pixels in the image and the pattern of the image [3]. Therefore the nodes with subimages that have many partial segments of connected components will have a higher load, as there are more entries in the resolution table to be resolved by the path resetting algorithm. In the case of images such as the ones illustrated in figures 5a and 5c, the nodes will have the same amount of load due to the symmetry of the images except for the leftmost and rightmost node in the linear array of nodes.

First Algorithm				
Picture	Elapsed Time(ms)			
Snake	1546			
Emblem	2962			
Spiral	1859			
ID	3215			

Table 2. The execution time of the initial algorithm on 32 nodes.

Tables 1, 2 and 3 provide the elapsed execution time of the code consisting of the sequential and the conflict resolution algorithms.

The results of the alternative algorithm for conflict resolution show a considerable improvement over the initial algorithm specially in the case of Emblem and ID where the elapsed time using the alternative algorithm is less than half of that of the initial algorithm. The disadvantage of the initial algorithm is due to the fact that for every new pair (l1, l2) added to the resolution table all (l3, l1) entries added in the preceding recursion levels must be reset to (l3, l2). Whereas in the case of the alternative algorithm resetting a set of previously resolved labels consist of performing a union operation on two disjoint sets which has a running time of O(1).

Second	Algorithm
Picture	Elapsed Time(ms)
Snake	1078
Emblem	1441
Spiral	1061
ID	1417

Table 3. The execution time of the alternative algorithm on 32 nodes.

There is certain amount of overhead involved with the load balancing technique we have used. As the boundary-overlap resolution algorithm is applied to each node in a group of nodes in a vertical slice, a resolution table is produced. These intermediary resolution tables must be merged to result a final resolution table. Distributing the boundary pixel arrays and merging the intermediary resolution tables are the major overhead factors responsible for the nonlinearity of boundary-overlap resolution running time with respect to the number of nodes partaking in the process of boundary resolution.

Load Balancing				
Picture	Elapsed Time(ms)			
Snake	1042			
Emblem	1356			
Spiral	1017			
ID	1370			

Table 4. The execution time of the final algorithm with load balancing on 32 nodes.

7. Future Research

Although the load balancing technique we have introduced does show an improvement to the conflict resolution algorithm, the overhead is still considerable and as the number of nodes increases the elapsed time for the conflict resolution becomes comparable with the execution time of the sequential algorithm. We are currently concentrating on reducing the overhead associated with the load balancing technique.

8. References

- [1] Kak, Avanish C., Azriel Rosenfeld.

 Digital Picture Processing. New York:

 Academic Press, 1982.
- [2] Reingold, Edward M., Jurg Nievergelt and Narsingh Deo. Combinatorial Algorithms: Theory and Pracice, Englewood Cliffs: Prentice Hall, 1977.
- [3] Tarjan, R. E., J. Van Leeuwen.

 Worst-Case analysis of set union algorithms,
 J. ACM, 31 (1984).

Digital Halftoning by Parallel Simulation of Neural Networks

Robert M. Geist
Roy P. Pargas
Prashant K. Khambekar
Department of Computer Science
Clemson University
Clemson, South Carolina 29634-1906

Abstract

The digital halftone resolution problem may be stated as follows: given an $n \times n$ array V of real numbers, $V_{i,j} \in [0,1]$, produce an $n \times n$ array ω of binary integers, $\omega_{i,j} \in \{0,1\}$, such that ω , when displayed on a binary output device such as a computer monitor or laser printer, is a "good" representation of the real information, the intensities contained in V. Standard halftone resolution algorithms, such as ordered dither, often mask specular information contained in image data. A new algorithm, based on feedback neural networks, is described in detail and shown to provide an enhanced specular information display. Three parallel implementations and one parallel/vector implementation on an Intel iPSC/2 hypercube are described. The programs are run on two images, one of size (256×256) , and the other of size (1024 × 1024). Parallelism results in a speedup of 7.6, with an efficiency of 95%, using eight processors. Vectorization provides a time improvement of approximately 2.5 over nonvector implementations.

keywords: digital halftone, neural networks, fixed-point iteration, parallel simulation, iPSC/2 hypercube

1. Introduction.

The digital halftone resolution problem may be stated as follows: given an $n \times n$ array V of real numbers, $V_{i,j} \in [0,1]$, produce an $n \times n$ array ω of binary integers, $\omega_{i,j} \in \{0,1\}$, such that ω , when displayed on a binary output device such as a computer monitor or laser printer, is a "good" representation of the real information, the intensities contained in V. The obvious resolution algorithm, round the values in V, fails to satisfy most interpretations of "good". For instance, if $V_{i,j} = .4999999$ for all i, j, then $\omega_{i,j} = 0$ for all i, j, and a desired gray image is displayed as white. Consideration of neighborhood intensities seems imperative.

Many halftone resolution algorithms have been proposed (see [8]). The most commonly used is probably the ordered dither [2], in which we tile the image matrix V with a smaller fixed array D of threshold values, and then turn on the pixel (set $\omega_{i,j}=1$) if and only if $V_{i,j}$ exceeds the corresponding threshold value. A standard

4×4 tile is

$$D = \begin{bmatrix} 1/32 & 17/32 & 5/32 & 21/32 \\ 25/32 & 9/32 & 29/32 & 13/32 \\ 7/32 & 23/32 & 3/32 & 19/32 \\ 31/32 & 15/32 & 27/32 & 11/32 \end{bmatrix}$$

Note that a uniform intensity of 0.5 would cause 8 of every 16 pixels (every other one) to be turned on.

In Figure 1 we show a 1024×1024 pixel image of a ray-traced scene containing two spheres, a checked floor, three walls, and a rectangular mirror (on the back wall). This image was resolved using ordered dither D, and was printed on a conventional 300 pixel per inch laser printer.

Although this image offers reasonable shading, we contend that the ordered dither, as well as other commonly used halftoning algorithms, can mask much of the specular information available in the data. For this reason we have developed an alternative algorithm whose primary purpose is enhanced specular information display.

In section 2 we provide the theoretical foundations of this algorithm, and in section 3 we describe its implementation on an Intel iPSC/2 hypercube. Section 4 contains conclusions and current directions.

2. Algorithm Design.

Our algorithm is based on feedback neural networks [4, 6]. A neural network is collection of simple analog processing elements designed to mimic biological neurons. The computational paradigm provided by such networks is a radical departure from that of the classical von Neumann architecture. The "input" to such a network is a matrix of interconnections among the processing elements, together with an initial voltage that is applied to each element. The networks are designed so that the stable output voltages of the analog elements are binary. The collection of all binary output levels is then interpreted as the "result" of the computation.

A four element example, from [9], is shown in Figure 2. Neurons are represented by amplifiers, each providing both standard and inverted outputs (voltage

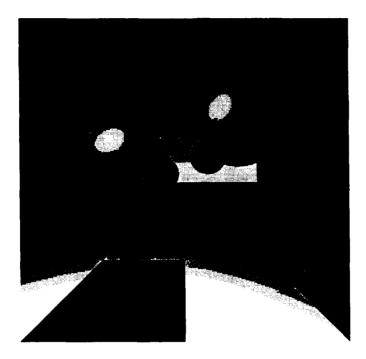


Figure 1: Ray-traced image resolved by ordered dither.

 $\theta_i \in [-1,1]$). Synapses are represented by the physical connections between input lines to the amplifiers and, in feedback, output lines from the amplifiers. Resistors are used to make these connections. If the input to amplifier i is connected to the output of amplifier j by a resistor with value R_{ij} , then the conductance of the connection is T_{ij} , whose magnitude is $1/R_{ij}$ and whose sign is determined by whether the connection to amplifier j is from the standard or inverted output. Hopfield [6] showed that when the matrix T is symmetric with zero diagonal and the amplifiers are operated in "high-gain" mode, the stable states of the network are binary ($\{-1,1\}$) and are the local minima of the computational energy,

$$E(\theta) = (-1/2) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} T_{ij} \theta_i \theta_j - \sum_{i=0}^{N-1} \theta_i I_i. \quad (1)$$

Here I_i is the external input to amplifier i.

Such neural networks offer a natural representation of a binary display. Each pixel is represented by a neuron that is connected to and influenced by its neighboring pixels (neurons). A network binary stable state is then a halftone resolution.

Several authors have considered applications of neural nets to digital halftoning [1, 3], but difficulties remain. First, there is no natural mapping from the desired image intensities, the V matrix, to the network parameters, the T_{ij} 's and the I_i 's. Reasonable choices abound. In our implementation we have selected a simple but easily motivated specification for these values. If

we scale the intensities over [-1,1] by letting $v_i = 2V_i - 1$, then our choice is given by

$$I_i = v_i - C \sum_{j \in nhbd(i)} v_j$$

$$T_{ij} = -K_{i,j}(2 - |v_i + v_j|)$$

where C and $K_{i,j}$ are non-negative constants. The term $K_{i,j}$ depends only upon the mod 4 row and column numbers of pixels i and j, and is symmetric in i and j.

Note that as v_i (and hence I_i) becomes larger, it becomes more important to turn that pixel on (set $\theta_i = 1$) to reduce E in (1). However, there are some attenuating factors. From (1) we see that $(-1/2)T_{i,j}$ can be viewed as the strength with which we insist that adjacent pixels assume opposite parity, and this is at a maximum when the underlying intensities are of equal magnitude and opposite sign, e.g. one black and one white $(v_i = 1, v_i = -1)$ or both gray $(v_i = v_j = 0)$.

We should also note that some control over average region intensity is at our disposal. If we let $m = \lfloor \sum_{i \in R} V_i + 0.5 \rfloor$ denote the rounded total intensity over region R, then we can add to $E(\theta)$ a summand of the form

$$C(\sum_{i\in R}\frac{\theta_i+1}{2}-m)^2$$

where C > 0, and, to maintain a zero-diagonal T matrix, another of the form

$$C(\sum_{i\in R}\frac{1-\theta_i^2}{4}).$$

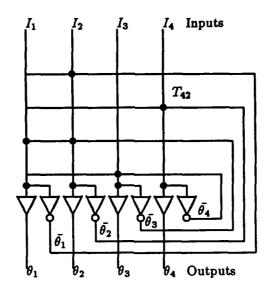


Figure 2: Four element feedback network.

The result is equivalent to adding C(m-|R|/2) to each I_i and -C/2 to each $T_{i,j}$ $(i,j \in R)$, and the net effect is to force m of |R| pixels on. Applied on a global scale, this modification lends force to providing a resolution with correct average intensity.

A second difficulty we face is that networks of $N = 1024^2 = 1,048,576$ neurons have not yet been built, and we must resort to net simulation, which can require excessive memory space and execution time.

Net simulation is traditionally approached (e.g. [9]) as a numerical integration of the system of N differential equations describing the operation of the amplifiers [6]:

$$C_i du_i/dt = \sum_i T_{ij} g(u_j) - u_i/R_i + I_i.$$
 (2)

Here the u_i are internal input voltages to the amplifiers, and are related to the desired output voltages, the θ_i , by a sigmoidal gain function, g(x). A reasonable choice for g(x) is a scaled hyperbolic tangent, $g(x) = \tanh(\lambda x)$. Here λ is called the gain. The C_i are the input capacitances of the amplifiers, and $R_i = 1/(1/\rho + \sum_j |T_{i,j}|)$, where ρ is amplifier input resistance.

We have found numerical integration of large (2²⁰ neuron) systems of the form (2) to be extremely time consuming, and therefore have developed an alternative approach. Any equilibrium of (2) is given by

$$0 = \sum_{j} T_{i,j} g(u_j) - u_i / R_i + I_i$$

that is,

$$u_i = R_i(\sum_j T_{i,j}g(u_j) + I_i)$$

or simply

$$u = G(u),$$

where G(u) = diag(R)(Tg(u) + I), diag(R) has R_i 's on the diagonal and 0's elsewhere, and $g(u) = (g(u_1), g(u_2), ...)$. Thus we seek a fixed point of a certain N-dimensional function. If | | denotes the max norm on Euclidean N-space and | | | its induced matrix norm, then since $R_i < 1/\sum_j |T_{i,j}|$ we have

$$|G(u) - G(u')| = |diag(R)T(g(u) - g(u'))|$$

$$\leq ||diag(R)T|| \cdot |g(u) - g(u')|$$

$$< |g(u) - g(u')|$$

$$\leq \lambda |u - u'|,$$

where the last inequality follows from a Taylor expansion and bounded derivative of g. Thus, for gain $\lambda < 1$, convergence of the simple iteration scheme, $u^{k+1} = G(u^k)$, is straightforward (see, e.g. [7]). Unfortunately, the Hopfield result speaks only of high-gain operation, and we must consider $\lambda > 1$, where the simple iteration is likely to diverge. Fortunately, there is an intriguing alternative.

In [5] Hillam established a remarkable result for functions on the real line: if $f:[a,b] \to [a,b]$ satisfies $|f(x)-f(y)| \leq M|x-y|$, then the iteration scheme

$$x_{n+1} = \frac{1}{M+1}f(x_n) + \frac{M}{M+1}x_n \tag{3}$$

converges to a fixed point of f. To our knowledge, the conjecture that this result extends to higher dimensions remains unresolved.

Nevertheless, we have found substantial empirical evidence to support it. Using (3) with $M=\lambda$, we find that convergence to a fixed point of G (that is, average component error $|u_i - G(u_i)| < 10^{-10}$) usually requires fewer than 200 iterations. We have not found a net for which this scheme fails to converge.

In Figure 3 we show the results of application of this algorithm to the same data used in Figure 1. We note the substantial addition of specular information. Spheres contain reflected images of the floor, the side walls (including the extent of the walls), and even each other. The walls also contain marked reflections of the floor and the mirror.

3. Implementation.

A sequential iteration on a floating point vector, u, of length 1,048,576 can represent an enormous computational expense, and a parallel implementation is highly desirable. We implemented our net simulation algorithm on an Intel iPSC/2 hypercube with 16 nodes, each with Intel 80386/80387 scalar processors and 8 megabytes of memory. Eight of the hypercube nodes have vector processors and an additional megabyte of vector memory.

. The program was written in C and run on one image of size 256×256 (Figure 4) and one of size 1024×1024 (Figure 3). Timing results of these runs are summarized in Table 1.

Four versions of the algorithm were developed: a sequential version (S), a parallel version (P1) running on eight scalar nodes, a second parallel version (P2) on eight nodes, a parallel version (P3) on sixteen nodes, and a parallel/vector version (PV) running on eight vector nodes. The large (1024×1024) image (Figure 3) was produced using P3. All other programs used the the smaller (256×256) image (Figure 4) for data. For P1, P2, and PV, input data are distributed evenly among eight or sixteen nodes, a node receiving an equal number of rows of input data. Program S, of course, holds all the data in a single node's memory.

Programs S and P1 were versions designed to conserve memory. Hence, the array containing the pixel intensities were updated in place, and functions (such as g(u)) were reevaluated whenever needed (rather than saved in temporary arrays). There are two advantages to this approach: (a) less memory is used since additional arrays to hold new values temporarily are not required, and (b) the method converges with fewer iterations since new updated values are used immediately. Two disadvantages however are: (a) vectorization of the operations is difficult, and (b) the algorithm requires multiple eval-

uations of the (computationally costly) function g(u).

Programs P2 and PV update each pixel's intensity using only old values of its neighbor's intensities. Separate arrays hold the new u and g(u) values. This requires more memory per node and more iterations (compare P1 and P2). However, because g(u) is evaluated only once for each new pixel intensity u, total number of operations and overall execution time are less for P2. PV is a vectorized version of P2. Additional memory is used by PV to make vectorization more efficient. Vectorization is possible because of the use of only old values of u to compute new values.

Note that each complete intermediate iterate, u^{k+1} , can be regarded as a halftone resolution and displayed, thus allowing us to observe the convergence. In Figure 4 we show intermediate resolutions of the 256×256 image (a digitized photograph) at iterations k = 4, 8, 12, 16, 24, 28, 32, 50, 100.

The ratio of execution times of versions S and P1 shows a speedup, due to eight-way parallelism, of 6080/800 = 7.6, with an efficiency of 7.6/8 = 95%. Comparing PV and P2, the executions times show that the improvement factor due to vectorization is approximately 289/114 = 2.5.

Table 1, column P3, gives the time required to generate the 1024×1024 pixel image shown in Figure 3. The method, like P2 and PV, computes the new u matrix using old u values. This version is not vectorized because vectors were available only on eight of the sixteen nodes of the iPSC/2 used. It required approximately 176 iterations taking 2579 seconds, or 43.3 minutes, of computation, and 110 Mbytes of memory. Had vectors been available on all nodes, we estimate that a vectorized version would improve by a factor of 2.5, or require $2579/2.5 \approx 1032$ seconds or 17.3 minutes. To the best of our knowledge, Figure 3 is the only neural network-generated image of this size.

4. Future Work.

We are currently working on a vectorized version of P3. We are modifying program PV in an attempt to reduce memory requirements. The goal is to fit the data into the memory available on eight vector nodes. A parallel/vector version should reduce the execution time by a factor of ≈ 2.5 , i.e., down to approximately 34.6 minutes using eight vector nodes. The ability to view a new image every 35 minutes should greatly facilitate the user's quest for the parameters that generate the most accurate images. We also continue to experiment with different parameters K and C, to discover relationships between the parameters and the quality of specular information obtained. Finally, we are currently experimenting with variations of the neural network algorithm used.

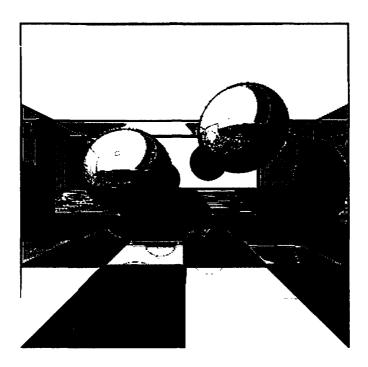


Figure 3: Ray-traced image with enhanced specular information.

Version	S	P1	P2	PV	P3
Number of Nodes	1	8	8	8	16
Image Size (N)	256	256	256	256	1024
Execution Time (secs)	6080	800	289	114	2579
Memory Required (Kbytes per node)	5720	755	892	960	6877
Number of Iterations	133	134	154	154	176
Number of Operations (millions)	641	641	319	319	5810
MFLOPS	0.105	0.801	1.105	2.809	2.253

Table 1: Summary of results

References

- [1] D. Anastassiou. Neural net based digital halftoning of images. Proc. IEEE Int. Symp. on Circuits and Systems, June 1988.
- [2] J. Foley and A. Van Dam. Fundamentals of Interactive Computer Graphics. Addison-Wesley, Reading, Massachusetts, 1984.
- [3] R. Geist and R. Reynolds. The most likely steady state for large numbers of stochastic traveling salesmen. Proc. First Int. Workshop on the Numerical Solution of Markov Chains, pages 569-581, Raleigh, NC, January, 1990.

- [4] S. Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks*, 1:17-61, 1988.
- [5] B.P. Hillam. A generalization of krasnoselski's theorem on the real line. Math. Mag., 48:167-168, 1975.
- [6] J.J. Hopfield. Neurons with graded response have collective computational properties like those of twostate neurons. *Proc. Natl. Acad. Sci.*, 81:3088-3092, 1984.
- [7] E. Isaacson and H.B. Keller. Analysis of Numerical Methods. John Wiley & Sons, New York, 1966.
- [8] J. Jarvis, C. Judice, and W. Ninke. A survey of techniques for the display of continuous tone pictures on bilevel displays. Comp. Graphics Image Process., 5:13-40, 1976.



Figure 4: Intermediate Resolutions of a Digitized Photo.

[9] E. Page and G. Tagliarini. Algorithm development for neural networks. In *Proc. SPIE Symp. on In*novative Science and Technology. Los Angeles, CA, January 1988.

Hypercube Algorithm for Image Decomposition and Analysis in the Wavelet Representation

Terrance L. Huntsberger Intelligent Systems Laboratory Department of Computer Science Beverly A. Huntsberger
Parallel Supercomputer Initiative
Computer Services Division

University of South Carolina Columbia, SC 29208

Introduction

The tremendous amount of data contained in an image oftentimes precludes the extraction of useful information in real-time environments. A multiresolution representation can be used to obtain structural properties of a single image or sequences of images[8]. These structural properties are useful for such operations as texture analysis, image segmentation, object identification and stereo matching[7,12,11].

A number of methods have been suggested to obtain a multiresolution representation of images. The pyramidal representation of Burt[1] and Crowley[3]. the morphological filter approach of Chen and Yan[2], the multiscale Gaussian filter of Marr and Poggio [10], the subband coding technique of Woods and O'Neil[14], and the wavelet representation of Mallat[9] are some examples. The wavelet representation offers the best basis for image analysis, since the orthonormal basis guarantees no correlation between image details sampled at different scales. Correlation found in the Burt and Crowley pyramid structure and the Chen and Yan morphological space hamper pattern recognition operations due to difficulties in selection of appropriate distance metrics. In addition, the wavelet representation has good localization properties in the Fourier and spatial domains[9].

This paper presents a hypercube algorithm for generating the wavelet representation of an image or sequence of images. All of the approximation and detail images of the representation can be obtained simultaneously. An experimental study is performed on images using the 1024 element NCUBE hypercube at the University of South Carolina. The next section gives a description of the development of the orthonormal wavelet representation for images. This is followed by details of the hypercube implementation. Finally, the

results of some experimental studies are shown.

Wavelet Representation

An image can be considered to be a member of the function space $\mathbf{L}^2(\mathbf{R}^2)$. Expansions of $\mathbf{L}^2(\mathbf{R}^n)$ functions are obtained from translations and dilations of the wavelet functions $\psi(x)[9]$, where n is the dimensionality of the vector space. The multiresolution approximation of an image is given by $A_{2j}f(x,y)$, where $j \leq 1$ and f(x,y) is the two-dimensional gray-scale image. The operator A_{2j} projects the image on a vector space $\mathbf{V}_{2j} \subset \mathbf{L}^2(\mathbf{R}^2)$. This projection space has the properties of causality, translation invariance and convergence to the original signal as $j \to +\infty$.

The orthonormal basis of V_{2^j} is derived from a scaling function $\phi(x, y)$, whose Fourier transform is a low pass filter for images. This means that the operator A_{2^j} is equivalent to a low pass filtering of the image followed by a uniform sampling at the resolution 2^j . If the Fourier transform of the scaling function is given by:

$$\hat{\Phi}(\omega,\upsilon) = \prod_{p=1}^{+\infty} \prod_{q=1}^{+\infty} H(2^{-p}\omega, 2^{-q}\upsilon),$$

where

$$\tilde{H}(\omega, v) = \sum_{n=-\infty}^{+\infty} \sum_{k=-\infty}^{+\infty} h(-n, -k) e^{i(n\omega+kv)},$$

then A_{2} , f(x,y) can be found by convolving the image with \tilde{H} and keeping every other row or column in a two pass algorithm. The convolution filter \tilde{H} is separable which makes it suitable for a parallel implementation. Since the function is separable $\Phi(x,y)$ can

be decomposed as:

$$\Phi(x,y) = \phi(x)\phi(y).$$

The discrete spatial form of the $\phi(x)$ scaling function can be found using a truncated series approximation. If the series is truncated at n=4, a convolution filter of size 1x9 is generated. This filter is shown in Figure 1. An image decomposition on the hypercube will have to pass a constant number of four rows across processor boundaries in order to apply this filter.

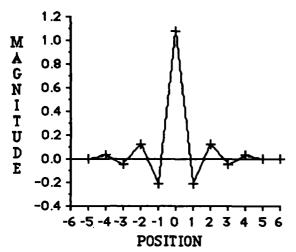


Figure 1 Ø(x) scaling function

The difference in information between the image at resolution 2^{j+1} and 2^j is called the detail signal. This detail signal is given by the orthogonal projection of the original signal on the orthogonal complement of V_{2^j} in $V_{2^{j+1}}$. The orthonormal basis for the detail signal is given by O_{2^j} and is generated by scaling the wavelet $\Phi(x,y)$ by 2^j and translating to a grid with a proportional spacing of 2^{-j} . The detail signal $D_{2^j}f(x,y)$ is obtained by convolving the image with \tilde{G} , where \tilde{G} is the mirror filter of \tilde{H} , followed by keeping every other row or column once again in a two pass algorithm. The mirror filter basis is related through the expansion coefficients of \tilde{H} as

$$g(n) = (-1)^{1-n}h(1-n).$$

The wavelet associated with \tilde{G} is generated using:

$$\hat{\psi}(\omega) = G(\frac{\omega}{2}) \hat{\phi}(\frac{\omega}{2}).$$

Application of this wavelet to an image is equivalent to a band pass filter centered at the origin which passes signals in the frequency range of -2π to $-\pi$ and π to 2π . The spatial version of this wavelet $\psi(x)$ is shown

in Figure 2. The original discrete image f(x,y) can be reconstructed using the information contained in the approximation signal $A_{2^{-J}}f$ and the three detail signals $(D_{2^{j}}f), -J \leq j \leq -1$.

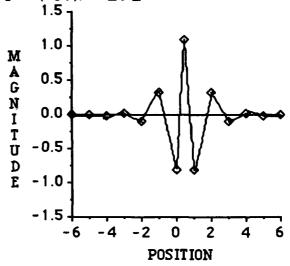


Figure 2 $\psi(x)$ scaling function

The next section contains details of the hypercube implementation of the algorithm.

Hypercube Wavelet Decomposition

The scaling function $\Phi(x, y)$ can be decomposed into the product $\phi(x)\phi(y)$. The same operation can be performed on the wavelets, except now instead of a single wavelet as there was in the one-dimensional case, there are now three. These three are given by:

$$\Psi^1(x,y) = \phi(x)\psi(y)$$

 $\Psi^2(x,y) = \psi(x)\phi(y)$
 $\Psi^3(x,y) = \psi(x)\psi(y).$

This form of the wavelets leads to three detail signals given by $D_{2j}^1 f(x,y)$, $D_{2j}^2 f(x,y)$ and $D_{2j}^3 f(x,y)$. The horizontal, vertical and corner structure of the image can be derived from these detail signals.

Since the wavelet transformation is separable, operations on the rows can be done followed by operations on the columns. In addition, the convolution with \tilde{H} can be done in parallel with the convolution using \tilde{G} . The following five step algorithm will generate A_{2^j} , $D^1_{2^j}$, $D^2_{2^j}$ and $D^3_{2^j}$ from the original unscaled image $A_{2^{j+1}}f(x,y)$.

HYPERCUBE WAVELET ALGORITHM

- Step 1: Subdivide image into N/4 strips, where N is the total number of processors. Distribute these strips to the processors such that the equivalent of four full images have been sent.
- Step 2: Convolve the rows of two of the distributed images with \tilde{G} , and the rows of the other two distributed images with \tilde{H} .
- Step 3: Sample resulting images by throwing out every other column.
- Step 4: Convolve the columns of two of the images from Step 3 with \tilde{G} , and the columns of the other two distributed images with \tilde{H} .
- Step 5: Sample resulting images by throwing out every other row.

The algorithm given above is optimal in the sense of communication in that only shared rows/columns need be communicated between nodes. A recent study done by Jones et al has indicated that a rectangular decomposition with the number of rows a factor of four times the number of columns is the optimal image decomposition for the NCUBE system[6].

Experimental Studies

In order to test the efficiency of the hypercube mapping we applied the algorithm to an image of the robot test area at ORNL. The image resolution was 256 rows by 256 columns with 256 levels of gray. The image was mapped to the hypercube using a ring mapping since at most only four rows of information needed to be transferred between processors. If the filter size is enlarged beyond 1x9, the added image information would have to be transferred. The original image is shown in Figure 3.

The hypercube algorithm described above allows the approximation signal and all of the detail signals to be derived simultaneously. Results of the application of the algorithm to the image in Figure 3 are shown in Figure 4. The display format used here is to place the approximation image in the upper left hand window, horizontal component image in the upper right window, vertical component image in the lower left window and the corner component image in the lower right hand window in the Figure. Details of the horizontal and vertical edges in the machine cabinets are preserved, while at the same time the diagonal stripes on the floor are apparent in all of the detail images.

Sensitivity to noise is evident in the center of the corner image in the lower right hand window of Figure 4.



Figure 3 Original image

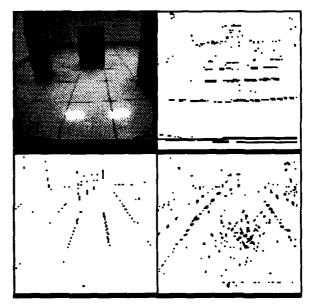


Figure 4 Wavelet decomposition

Since the communication is of constant size, efficiency of the algorithm should be almost constant. Efficiency is plotted versus dimension of the subcube for the image used in the study in Figure 5. Up to a size seven subcube communication will only be with

nearest neighbors in the ring. However, since each processor only contains two rows in the size seven subcube mapping, two transfers are necessary to get the

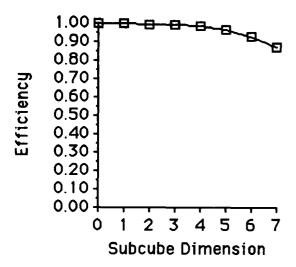


Figure 5 Algorithm performance

needed four rows of information. This effect is apparent in the size seven subcube efficiency of 87%. The total time taken for the wavelet transform at this resolution subcube was 89.7 milliseconds, giving an equivalent double precision floating point rate for all four of the wavelet image generations of 54.8 Megaflops. This timing bench includes the row transfers as well as the type casting from char to double and double to char necessary due to memory limitations on the nodes.

Conclusions

We have presented a parallel version of the wavelet decomposition algorithm for images. The algorithm exploits the SPMD nature of the NCUBE hypercube and allows further pattern recognition to be done with the horizontal, vertical and corner information being resident on the nodes. Efficiency of the algorithm is quite good up to a size seven subcube, with a performance of 87% at that dimension. The degradation in performance at this resolution would be offset by a larger image, since more rows would be contained within the memory of a single processor. Since the algorithm was designed for SPMD mode, it should port quite easily to a SIMD machine. The port to the SIMD MP-1 machine manufactured by MasPar Corporation was straight-forward and the results of timing runs indicate a performance of 108 double precision Megaflops on a 4096 node version. The execution time was 31.7

milliseconds, which indicates that the algorithm can be used for real-time image analysis.

We are presently extending the analysis to a three dimensional wavelet decomposition of a sequence of images, where the third dimension is time. Motion analysis using region features has been previously examined at the full image scale[13], and the wavelet representation offers the possibility of a parallel examination of the time evolution of multiple features derived from a segmentation analysis using a parallel self-organizing feature map[5]. In addition, Mallat has shown that texture analysis can be done with the wavelet representation using the fractal dimension derived from the power function spectra[9]. This type of analysis can be merged with the fractal signature approach[4].

Acknowledgements

The authors would like to thank Dr. Bert Still of the Parallel Supercomputer Initiative at the University of South Carolina for numerous discussions on the wavelet transform theory. In addition, Judson Jones of ORNL deserves special mention for providing the image and image processing framework for this research. We would also like to thank Jeff Fier of MasPar Corporation for the porting of the algorithm and timing runs on the MP-1 SIMD architecture.

References

- [1] P. J. Burt and E. H. Adelson, "The Laplacian pyramid as a compact image code". *IEEE Trans. Commun.*, Vol. 31, 1983, pp. 532-540.
- [2] M. Chen and P. Yan, "A multisclaing approach based on morphological filtering". *IEEE Trans.* PAMI, Vol. 11, 1989, pp. 694-700.
- [3] J. Crowley, "A representation for visual information". Robotic Inst. CMU, Tech. Rep. CMU-RI-TR-82-7, 1987.
- [4] B.A. Huntsberger and T. L. Huntsberger, "Hypercube algorithms for multi-spectral texture analysis". *Proc. HCCA4*, Monterey, CA, Mar 1989.
- [5] T. L. Huntsberger and P. Ajjimarangsee, "Parallel self-organizing feature maps for unsupervised pattern recognition". Int. Journ. General Systems, Spec. Issue Adv. in Patt. Recog., in press.

- [6] J. P. Jones, M. Beckerman and R. C. Mann, "Concurrent implementation of two multisensor integration algorithms for mobile robots". Proc. SPIE Conf. Sensor Fusion II: Human and Mach. Strategies, Philadelphia, PA, Nov 1989.
- [7] B. Julesz, "Textons, the elements of texture perception and their interactions". Nature, Vol. 290, 1981.
- [8] J. Koenderink, "The structure of images". Biological Cybernetics, Springer-Verlag, New York, 1984.
- [9] S.G. Mallat, "A theory for multiresolution signal decomposition: The wavelet representation". IEEE Trans. PAMI, Vol. 11, 1989, pp. 674-693
- [10] D. Marr, Vision, Freeman Press, New York, 1982.
- [11] D. Marr and T. Poggio, "A theory of human stereo vision". *Proc. Royal Soc. London*", Vol. B204, 1979, pp. 301-328.
- [12] A. Rosenfeld and M. Thurston, "Coarse-fine template matching". IEEE Trans. Syst., Man, Cybern., Vol. 7, 1977, pp. 104-107.
- [13] Y. Soh and T. L. Huntsberger, "Hypercube algorithms for dynamic scene analysis". *Proc. HCCA4*, Monterey, CA, Mar 1989.
- [14] J. Woods and S. O'Neil, "Subband coding of images". IEEE Trans. ASSP, Vol. 34, 1986, pp. 1278-1288.

Parallel Processing Applied to 3D Coronary Arteriography

Alok Sarwal Jayashree Ramanathan

Computing Environments Group Universal Energy Systems, Inc. 5162 Blazer Memorial Parkway Dublin, OH 43017

Abstract

One important reason for the lack of acceptance of coronary arteriography techniques is the four hours (approximate) required for accurate 3D quantification of an arterial tree, thus making it prohibitive for the clinician to utilize. Parallel processing techniques can greatly speed up the image processing and analysis of 3D arterial trees. It has been demonstrated in this project that the reconstruction of the three dimensional image and arteriographic measurements can be made close to real time using these techniques.

1. Problem Description

Coronary arteriography is currently the standard technique for evaluating the condition of the coronary arteries. This procedure not only determines the need for revascularization[1], but also the degree of success of surgery involving angioplasty and bypass grafting. Currently, coronary artery reconstruction is performed on several sets of Digital Substraction Angiography images from different patients. In each case, the images are obtained from a Siemens Angioscope D interfaced to a Digitron II digital image acquisition system. This system can currently acquire 512 by 512 pixel images at 30 frames per second simultaneously with conventional cine film acquisition. Reconstruction requires at least two views of known orientation. For each of the twoview images, the arteries are opacified with an iodonated contrast medium, as a sequence of x-ray images are acquired.

The flow of information is shown in Figure 1. The 3D reconstruction requires that the view geometries be known as accurately as possible. Corrections due to geometric misalignment[2], x-ray scattering, artifacts etc. have to be made before

Dennis L. Parker Jiang Wu

Department of Medical Informatics LDS Hospital 325 Eighth Avenue Salt Lake City, UT 84143

accurate view geometries are realized. The first step for obtaining these geometries is segmentation of the arteries. This requires an operator to specify the branch points and other recognizable points, also referred to as the node points. This interaction establishes the original hierarchy and the correspondence of node points between views. It also divides the original arterial tree into smaller artery segments or branches[4]. The next step is the determination of vessel centerline and edges of each segment in each view, which proceeds without operator intervention[6]. The 2D geometry coordinates of each segment are mapped onto the 3D reconstructed geometry[3,5,7]. The 3D reconstruction becomes more accurate as the number of views of x-ray angiograms are increased.

The above mentioned operations are very computationally expensive. Moreover as the number of views increases, the time complexity for the algorithm increases markedly. The implementation of a multi-processor based workstation utilizing parallel programming techniques will not only reduce the time for coronary reconstruction, but also make the three and four view reconstruction a reality for even more accurate results.

2. Description of Existing Algorithms

The existing sequential algorithms have been implemented on a VAX-780 computer to provide 3D reconstruction of coronary arteries. Input images are obtained from two-view ECG correlated x-ray angiograms. A target structure consisting of the node points is entered on the first of a sequence of images[3,4] in one view using the Digitron II. Automatic edge detection is used to detect the centerline and edges of the vessels[7]. The edge detection algorithm represents the computationally intensive part of the software and is used many times in the course of the reconstruction. Anglecorrected densitometric calculations are used to refine the vessel cross section[5]. reconstruction is completed by a distance minimizing point matching technique.

¹Phase 1 Small Business Innovation Research Project supported by National Institute of Health (NHLBI) under Contract # 1R43HL42208-01.

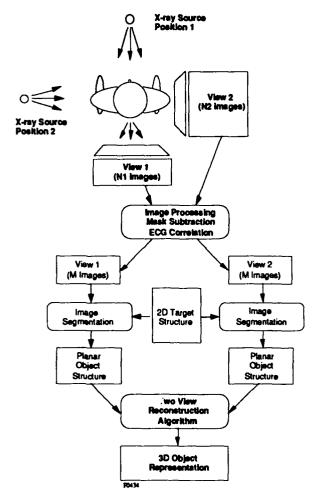


Figure 1: Information Flow in 3-D Coronary
Artery Reconstruction

Based on an operator entered target structure, vessel centerlines and edges are detected in an automated manner throughout the heart cycle in each view. This centerline and edge information in 2D is used to obtain the 3D representations of the arterial bed throughout the coronary cycle. The extraction of the image information into sub-image is performed orthogonal to the idealized search target such that the rows of the extracted matrix consist of image values along this orthogonal path. Edge features are then enhanced using matched filter convolution to create a likelihood matrix. The dynamic search software module is then used to map the optimal global path through this matrix. This technique is applied on a segment or a portion of the entire image and is repeated to process all the other segments in the image. Typically each view will have about 20 segments. The resultant datastructure consists of a set of 2D edges. The

densitometric cross section measurements are obtained corresponding to each point along the vessel segment.

The 3D reconstruction is obtained from each view using geometry information of the 2D arterial tree or plane tree as shown in Figure 1. For two or more views the problem is reduced to finding the intersection of a projection line representing the x-ray path for each element. From the 3D centerline the orientation of each vessel segment relative to each projection is computed. The area of cross section is computed for all elements of all segments using the orientation corrected plane tree measurements. Flow characteristics of the blood are obtained from transit time measurements of the leading edge of the iodine bolus passing through the artery bed. The reconstructed image can be displayed on a high resolution graphics monitor.

3. Analysis of the Computation-Intensive Routines

The convolution and dynamic search algorithms are invoked multiple times during the course of the complete 3D reconstruction. The convolution operation is invoked each time edge or centerline detection has to be performed. The edges and vessel centerline in 2D have to be calculated before any 3D reconstruction can be done. The dynamic search algorithm is also called as frequently as the convolution software. For a two-view reconstruction of the coronary artery structure the number of branches to be calculated will be about 1200 per second of x-ray image data, assuming that a total of 30 x-ray images are taken per second. For each such branch to be represented geometrically, the convolution and dynamic search has to be applied once for each edge and once for the centerline. There are instances where the operator may decide to recalculate the vessel geometry as it may need further refining These algorithms could be called a total of 3600 times for a total global reconstruction of the plane tree for one heart cycle. It is quite important that the above mentioned software is speeded up to provide an overall improvement in performance. complexity analysis of the above algorithms is examined for this purpose.

Time complexity of sequential convolution

$$T_1(t) = O(n*m*k)$$

where n is the number of rows and m is the number of columns of the extracted matrix, and k is the number of elements in the filter (kernel).

Time complexity of sequential dynamic search

$$T_2(t) = O(n*m*s)$$

where s is the step size for the search window for dynamic search.

4. Implementation on the Multiprocessor Workstation

The Transputer based system consists of multiple processing nodes arranged in a hypercube network. The processor in this case is the INMOS T-800 Transputer [8,9]. These processors, available on a card as a group of four, can be easily installed onto the IBM-AT bus. One very important reason for selecting the Transputer is that it provides one of the best cost vs. performance characteristic of any other distributed memory parallel processing system. It can also be easily added onto a desktop personal workstation. Two methods of parallelism were implemented in this feasibility study.

- The first is the fine grain method, where data partitioning is done for one branch of the extracted data to be processed at a time,
- The second is the coarse grain method, where there is segment parallelism and one segment being processed per processing node.

4.1. Fine Grain Method

The first and most important requirement for parallel processing is to map the algorithms for the problem to be solved onto each processor. The efficiency and speed up of the solution are enhanced sufficiently by selecting a good strategy for mapping this problem.

4.1.1. Mapping. The convolution algorithm convolutes a matched filter with elements of the extracted matrix. This operation performs the convolution row by row such that each element of the matched filter is multiplied with each element of each row of the extracted matrix. This algorithm can be parallelized to run on nonoverlapped subsets of data selected from the complete extracted matrix and then loaded on the various processors with no communication during the course of the convolution.

The only communication among processors is for the distribution and subsequent collection of the convolved sub-matrices. The partitioning of this data is done in the form of vertical strips as shown in Figure 2. The number of columns are divided equally among all the processors, and each has an almost equal number of elements of the matrix. Each processor then reads its partition of the complete matrix in its local memory to start with the convolution. At the end of the convolution the resultant matrices are available for parallel dynamic search. With this scheme of partitioning of the data, there is an almost linear speed up in these algorithms. The overhead of mapping is the matrix distribution before convolution and recombination of the sub-matrices after the dynamic search. This represents a fine grain approach to parallelism as applied to this particular problem.

Each processor will have a partition of size n rows by (m/p) columns, where p is the number of processors in the workstation.

Time complexity for the mapping of the data

$$T_3(t) = O(p)$$

Mapping for dynamic search can only be along vertical strips. In this case, the data partitions have to be a division of the number of columns only, because the search proceeds vertically and proper load balancing can only occur if each processor is kept busy with almost the same work load. The search starts at the last row and builds up one row at a time, till it reaches the first row. If the division had been along horizontal strips, then it would result in only one processor executing the search operation at one time and then passing its boundary path direction to its neighbor processor.

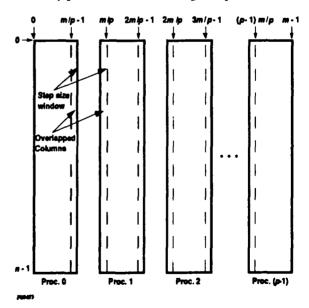


Figure 2: Vertical Strip Mapping for m by n columns on p processors.

4.1.2. Parallel Convolution. The extraction of image information into the sub-image is application dependent but shares the common feature that pixels are extracted orthogonal to an idealized search target. At the present time the initial target is a fixed radius circle and the subimage information is extracted orthogonal to this circle. The operator specifies a search target along which the extraction occurs. For coronary artery tracking the target is specified as a set of node points that are connected by segments. The individual sub-image matrices have to be convolved so that the image features are enhanced. The above mentioned convolution operation produces a Structure Likelihood Matrix (SLM). importance of the algorithm is in the structure enhancement of subtracted images using a 1D gradient density matched filter. The filter elements are application specific and are selected on the basis of the densitometric profile of the structure being tracked. One pass of the convolution algorithm enhances the features corresponding to one edge. The filter or kernel is reversed left to right for detection of the other edge. Finally a center finding filter is used for finding the vessel centerline.

The convolution algorithm can operate on a subset of the extracted sub-image data. The matrix is partitioned in equal parts so each processor can work on its partition independently. This parallel processing technique provides an overall speed up of this operation. In fact, the speed-up is quite linear and corresponds to the ideal maximum limit, over p processors where p > 1, as detailed in the results.

Time complexity of parallel convolution

$$T_A(t) = O(n^* m/p * k)$$

where (m/p) columns of the matrix will be convolved simultaneously

4.1.3. Parallel Dynamic Search. The Structure Likelihood Matrix (SLM) represents the enhanced image features for the extracted matrix. The higher the magnitude of the element of the SLM, the greater the probability of it being present in the image. The dynamic search algorithm finds the path through the SLM for the edge of a segment and builds up on greater number of such segments for a global solution. The path direction of each element in the bottom row is selected from a window of size equal to twice the path width. After the direction for each element in a row is selected, the magnitude of the element is added to the magnitude of element it points to, for an updated set of row elements. The path direction is

repeatedly calculated with these updated elements. The path is traced in this method for all the elements of each row for all the rows. At the end of this operation the top row will have the cumulative magnitude for the path traced for each element in that row, having started at the bottom row. The element with the largest magnitude in the top row will represent the most likely starting point of the path. The path traversed by this most likely element is traced by following the direction vectors through all the rows for an edge of the artery or organ.

The vertical strips are mapped onto the processors so that the computation envelope is an array of processors, one per partition, as shown in Figure 2. The step size or the search window provides the minimum and maximum limit for the direction vector of the path for an element. The elements at the boundary columns of the partitioned submatrices on a particular processor need to communicate with the neighboring processor, that has an overlap of it's search window. For example, assume that the element at column (m/p) and row (n-1) as shown in Figure 2, has a step size of 2 elements, such that it has a path extending 2 elements on the left and right directions respectively. As this element falls on a partition boundary, it will need to access two elements on the left adjacent columns which are in the partition loaded on processor 0. Likewise the right boundary elements of elements on processor 0 will have to access elements of its search window extending to the left hand columns of processor 1. Thus at each iteration there is the overhead of communication among processors. The total number of elements that have to be communicated to a neighbor at the end of each iteration of dynamic search are:

$$N_c = [n * (p - 2) * w]$$

where w is the path width of the search window.

4.2. Coarse Grain Method

An alternate method for parallel implementation is by adopting a coarse grain approach. The edge detection involving convolution and dynamic search operations is performed on multiple branches simultaneously. With this approach, p branches can be processed at the same time on each of the p processors, with one branch per processor. The branch image data corresponding to each segment is loaded on each processor, such that there is absolutely no interaction between processors during the convolution and dynamic search operations. At the end of the above operations the information relating to the two edges and the centerline of the vessel is stored in the data structure of the plane

tree, to be used in the 3D reconstruction operation. There are about 20 branches per view and so an array of about 40 processors can be utilized simultaneously for all the arterial tree data from two views.

The computation load on each processor is now a function of the size of the branch matrices. The processors are performing edge detection algorithms in an asynchronous manner. There can be the situation where one processor will complete the detection before another processor. This processor will not have to wait for computation on all other processors to be completed, if there is other computation like 3D geometry computation to be performed, else it has to let other processors catch up. Thus, the best performance can be expected in the condition that the number of branches are equal to the number of processors, and each branch matrix is of the same size.

5. Evaluation of Results

5.1. Fine Grain Method

The parallel convolution algorithm provides performance directly proportional to the number of the processors. There is a linear correlation between the number of processors and the relative speed up, as shown in Figure 3 and Figure 4. Relative speed up S can be defined as:

$$S = \frac{Timetakenon 1 \ processor}{Timetakenon \ p \ processors}$$

Figure 3 shows the results of the fine grain method. For parallel convolution in this method, there is no interaction among processors after the sub-matrices are loaded on the processor, and these matrices have no overlapped elements. Moreover the convolved matrices can be recombined quite inexpensively. Thus the perfect speed up for this algorithm is obtained during benchmark studies.

The parallel dynamic search has the overhead of communicating the elements corresponding to the path width at the boundaries of the sub-matrices. Typical path width w is 1 or 2 elements. The benchmark has been implemented for a path width of 2 elements. There is some amount of overlap between the communication and computation as applied to this case, such that as a processor is communicating to all its neighbors, it starts up the subsequent computation after some communication set up time. The performance for parallel dynamic search decays as this communication increases. The processors are used as a linear array, so there is

bi-directional communication of elements.

The fine grain method consists of the combination of parallel convolution and parallel dynamic search as applied to only one branch at a time. There is initial overhead in performing the mapping of the various partitions of data on the processors. Due to the most efficient method of partitioning selected for this method, there is a minimal overhead in comparison to the overall time for solution. This method provides a very good load balancing of the array of processors. The overall performance with this method is almost linear for up to 4 processors.

There is a slight decay in the speed-up characteristic curve at p = 4 because of the overhead in communication in dynamic search as shown in Figure 3. It is apparent that the performance is quite impressive and closely follows the theoretical limit. The analysis was performed for a test case of simulated edge data. The matrix was a 256 by 256 representation of a sinusoid like edge. The benchmark was calculated for 1, 2 and 4 processors respectively. It took 14, 7 and 4 seconds for the 1, 2 and 4 processor cases respectively. These timings do not include the time to read the matrix data from the hard disk-drive, which is 2 seconds... The results can be extrapolated for more processors for almost linear speed up. This method works well for large sized data sets, but performance diminishes as the partition size decreases. There is communication involved as each row of the matrix is searched. As more processors are added, the size of the partition will be small enough that the time for computation will be less than the time for communication and so the solution will be communication driven. At this point or close to this point, the performance will decrease and not be directly proportional to the number of processors.

5.2. Coarse Grain Method

This method applies convolution and dynamic search on a complete extracted matrix corresponding to a branch of the arterial tree. This method provides performance which is linear for a total of pbranches applied to p processors as shown in Figure 4. The x-ray data is segmented into unrelated subsets. The extraction of these submatrices and then the subsequent detection of edges for one branch is completely disjoint from the next, and so can be performed independently on different processors of the system. There could potentially be about 800 such segments to be processed over a complete coronary cycle. It is apparent that an 800 processor system can run on 800 segments to generate the geometry for a plane tree and then proceed with this coarse grain approach for 3D reconstructed set of branches.

The x-ray images were 256 rows by 256 columns of data. The arterial tree was segmented into submatrices for separate branches. The size of the largest extracted data matrix for a branch in this test was 122 rows by 60 columns, and the rest were smaller in size. The benchmark analysis was performed for a maximum of 4 processors and so 4 branches were selected from the segmented set of branches of the tree. The time taken for detecting both edges of this segment was 3 seconds. Thus the time for obtaining the edges for all 4 segments was 3 seconds. In comparison a single processor based computer system would have taken about 12 seconds to perform the same computation. As another case for observation, consider the computation for edge detection of 32 segments; the time taken by the coarse grain method will still be about 3 seconds, while the time taken on a single processor computer, with same processing power, will be about 96 seconds. There is a clear advantage in cost vs. performance in this approach, as applied to 3D coronary reconstruction.

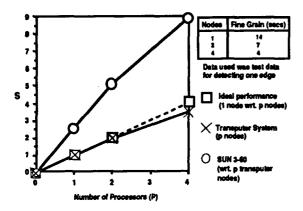


Figure 3: Fine Grain Method

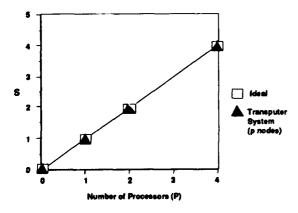
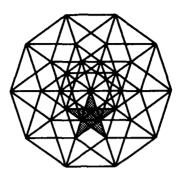


Figure 4: Coarse Grain Method

References

- [1] Ellis, S.G., Cain, K., Bourassa, M.G., Alderman, E.L., Lad lesion severity as a predictor of anterior infarction in CASS: visual vs. computer methodologies, Proceedings AHA (1985).
- [2] Jacques, P., Dibianca, F., Pizer, S., Kohout, F., Lifshitz, L., Delany, D., Quantitative digital fluorography computer vs. human estimation of vascular stenoses, Investigative Radiology 20, 45-52 (1985).
- [3] Parker, D.L., Wu, J., Pope, D.L., Bree, R.E., Caputo, G.R., Marshall, H.W., Three dimensional measurements of coronary arteries using multiview digital angiography, Rotterdam Conference on Quantitative Coronary Arteriography, (June 1987).
- [4] Parker, D.L., Pope D.L., Bree, R.E., Desai, R., Three dimensional reconstruction of vascular beds from digital angiographic projections, SPIE-Vol. 671, 50-59, International Workshop on Physics and Engineering of Computerized Multidimensional Imaging and Processing, (1986).
- [5] Parker, D.L., Pope D.L., Bree, R.E., Marshall H.: Three dimensional reconstruction and cross-section measurements of coronary arteries using ECG correlated digital coronary arteriography, Progress in Digital Angiocardiography, Ed., P.H. Heintzen, M.D., Martinus Nijhoff, Dordrecht, The Netherlands, (1987).
- [6] Pope, D.L., Parker, D.L., Gustafson, D.E., Clayton, P.D., Dynamic search algorithms in left ventricular border recognition and analysis of coronary arteries, IEEE Computers In Cardiology, 71-75, (1984).
- [7] Parker, D.L., Pope, D.L., White, K.S., Tarbox, L.R., Marshall, H., Three dimensional reconstruction of vascular beds, Proceedings of the Conference on Information Processing in Medical Imaging, Georgetown, (1985).
- [8] Jesshope, C., Reconfigurable transputer systems, 3rd. Conf. on Hypercube Concurrent Computers and Applications, 105-114, (1988).
- [9] Hey, A.J.G., Practical parallel processing with transputers, 3rd. Conf. on Hypercube Concurrent Computers and Applications, 115-121, (1988).



The Fifth Distributed Memory Computing Conference

7: Computer Vision

Surface Reconstruction and Discontinuity Detection: A Fast Hierarchical Approach on a Two-Dimensional Mesh

Roberto Battiti
105 Via Menguzzato, 38100 Trento, Italy
E-mail: battiti%genova.infn.it@iboinfn.bitnet

Abstract

Recently multigrid techniques have been proposed for solving low-level vision problems in optimal time (i.e. time proportional to the number of pixels). In the present work this method is extended to incorporate a discontinuity detection process cooperating with the smoothing phase on all scales. Activation of line element detectors that signal the presence of relevant discontinuities is based on information gathered from neighboring points at the same and different scales.

Because the required computation is local, parallelism can be profitably used. A mapping of the required data structure onto a two dimensional mesh of processors is suggested. *Domain decomposition* is shown to be efficient on MIMD computers capable of containing many individual cells in each processor.

Some examples of the proposed multiscale solution techniques are shown for two different applications. In the first case a surface is reconstructed from first derivative information (extracted from the intensity data), in the second case from noisy depth constraints.

1 Introduction

In the last years a sound scientific basis has been given to low and intermediate level vision that decodes information about three-dimensional surfaces and their properties.

Subsequent visual processing can be facilitated if the different constraints are transformed into a visible surface representation that unambiguously specifies surface shape at every image point.

It is practically very hard to recognize an object in a visual scene unless one knows how to choose the subset of evidence that derives from the same object. Hence discontinuities are necessary both to avoid washing away important information under the smoothness requirement, and to provide a primitive perceptual organization of the visual input into different elements loosely related to the human notion of objects. In some schemes the smoothing and discontinuity detection steps are done at different times, but there is a general suggestion that both should be done at the same

time, since the first is hiding evidence used by the second [11].

Psychophysics and practical implementations (see for example [1,6,10]) show that the early vision step can be done in parallel. Many computational units (neurons or processors) cooperate to reach the desired solution with a speed-up roughly proportional to their number.

In the following first the multigrid method with discontinuities is briefly described, second the parallelization strategy is outlined and finally some results are presented.

2 Multigrid Method with Discontinuities

Early vision can be considered "inverse optics", since its purpose is to undo the image formation process, recovering the properties of visible 3-D surfaces from the 2-D array of image intensities. In general the class of admissible solutions is restricted by introducing a priori knowledge. In the regularisation method the desired or plausible properties are enforced when the inversion problem is transformed into the minimization of a functional [12].

The stationary points of the functional are found by solving the Euler-Lagrange partial differential equations.

In standard methods for solving PDEs, the problem is first discretized on a finite dimensional approximation space. The very large algebraic system obtained is then solved using for example "relaxation" algorithms, which are local ¹ and iterative.

By the local nature of the relaxation process, solution errors on the scale of the solution grid step are corrected in a few iterations; on the contrary larger-scale errors are corrected very slowly. Intuitively, in order to correct them, information must be spread over a large scale by the "sluggish" neighbor-neighbor influence. A larger spread of influence per iteration demands large-scale connections for the processing units, i.e. a solution of the same problem on a coarser grid.

¹The local structure is essential for efficient use of parallel computation.

The pyramidal structure of the multigrid solution with $0 \le k \le L$, 0 = coarsest) is grids is illustrated in Figure 1.

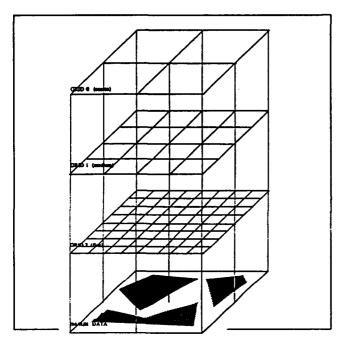


Figure 1: Pyramidal structure for multigrid algorithms.

This simple idea and its realization in the multi-grid algorithm not only leads to asymptotically optimal solution times (i.e. convergence in O(n) operations) but also dramatically decreases solution times for a variety of practical problems, as shown in [3].

In the multigrid strategy first relaxation is used to obtain an approximation with smooth error on a fine grid. Then, given the smoothness of the error, corrections to this approximation are calculated on a coarser grid, and, in order to do this, first relaxations are executed, then correction are calculated recursively on still coarser grids. The nested iteration scheme (use of coarser grids to provide a good starting point for finer grids) is used to speed up the initial part of the computation. Historically these ideas were developed starting from the sixties by Bakhvalov, Fedorenko and others (see Stüben et al. [13]).

It is shown in [3] that, with a few modifications in the basic algorithms, the actual solution (not the error) can be stored in each layer. This method is particularly useful for visual reconstruction, where we are interested not only in the finest scale result but also in the multiscale representation developed as a byproduct of the solution process. This is called full approximation storage algorithm and it is briefly described in what follows.

The algebraic system, obtained by discretizing the original problem on the different grids (numbered by k

$$\mathbf{A}^{h_k}\mathbf{z}^{h_k}=\mathbf{d}^{h_k}\tag{1}$$

The data on the finest grid define dhe, while for the coarser grids the right hand side dhe is obtained using the two extension (fine to coarse) and interpolation (coarse to fine) operators, respectively I^{\uparrow} and I^{\downarrow} in this

$$\mathbf{d}^{h_k} = \mathbf{A}^{h_k} \left(I^{\dagger} \mathbf{z}^{h_{k+1}} \right) + I^{\dagger} \left(\mathbf{d}^{h_{k+1}} - \mathbf{A}^{h_{k+1}} \mathbf{z}^{h_{k+1}} \right) \tag{2}$$

Before computation is begun on a grid finer than the current one, the initial values for z are updated as:

$$\mathbf{z}^{h_k} \longleftarrow \mathbf{z}^{h_k} + I^{\downarrow} \left(\mathbf{z}^{h_{k-1}} - I^{\uparrow} \mathbf{z}^{h_k} \right) \tag{3}$$

Instead, before computation is begun on a grid coarser than the current one, the initial values for z are updated as:

$$\mathbf{z}^{h_k} \longleftarrow I^{\dagger} \mathbf{z}^{h_{k+1}}$$
 (4)

The switching of control between different grids is explained in Figure 2

The sequential multigrid algorithm was used for solving PDE's associated with different early vision problems in [14], obtaining typical speed-up factors of 100.

Line Processes

Even if there are some results in the literature (see [15], [11,7]), up to now it is not clear how to combine in an optimal way the surface reconstruction and the discontinuity detection processes.

Various approaches are based on different degrees of cooperativity of the two processes, considering both time and scale (see [2] for details).

In some cases the discontinuity detection step is assigned to a separate preliminary process. Assuming this, in a regularisation approach the smoothness constraint will no more be enforced globally, but locally depending on the presence or absence of line processes.

In other schemes, discontinuities are detected after the smoothing step, for example by taking derivatives 3 and thresholding them appropriately.

²This definition agrees with the idea that coarse-scale corrections are a top - down influence. The definition given in "mathematical" texts is usually the opposite, so beware.

³Error in derivatives will be smaller after regularization.

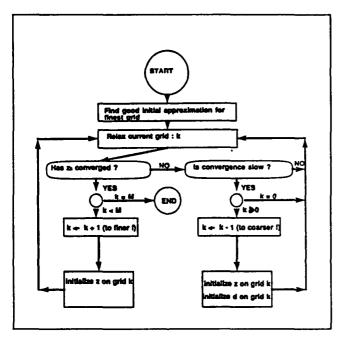


Figure 2: Flow of control in sequential multigrid (adapted from Brandt).

Finally other proposals consider cooperation of the two processes in time but do not consider the problem of organizing the cooperation in scale.

In [9] for example a new term is added to the energy function to favor a good discontinuity structure. In their hardware implementation, an analog network minimizes the "smoothness and data agreement energy" while, in a cyclic way, a digital network updates the line processes minimizing the "discontinuity energy".

Summarizing, in the first two approaches one process cannot make use of information exchange with the "dual" one, while in the last computation tends to be very slow for large images.

We propose to combine discontinuity detection and surface reconstruction in time and scale. To do this, we introduce line processes at different scales, interacting with neighboring depth processes (henceforth DPs) at the same scale and with neighboring line processes (henceforth LPs) on the finer and coarser scale. The reconstruction assigns equal priority to the two process types.

The recursive multiscale call mg(lay) is based on an alternation of relaxation steps and discontinuity detection steps as follows (in C language):

```
void mg(layer) int layer;
{
  int i;
```

```
if(layer==coarsest)step(layer);
else{
i=na;while(i--)step(layer);
i=nb;if(i!=0)
{up(layer);while(i--)mg(layer-1);down(layer-1);}
i=nc;while(i--)step(layer);
}

void step(layer) int layer;
{
  exchange_border_strip(layer);
  update_line_processes(layer);
  relax_depth_processes(layer);
}
```

Each step is preceded by an exchange of data on the border of the assigned domains, as explained in section 3 dedicated to the parallel implementation.

As we will show in the following this scheme not only greatly improves convergence speed (the typical multigrid effect) but also produces a more consistent reconstruction of the surface at different scales.

2.2 Mutual Interaction

During the course of the reconstruction, each LP updates its value in a manner depending on the values of other connected LPs in a neighborhood. It is useful to

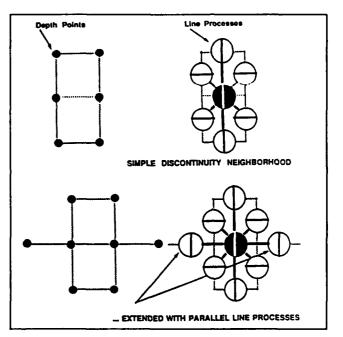


Figure 3: LP neighborhood inside a given layer.

define three different subsets of this neighborhood: the

set of connected LPs at the same scale SSN, its subset SSN* lacking the two parallel LPs (defined as the LPs at both sides of the given one and with the same orientation, see Figure 3) and the set DSN containing the connected LPs at the coarser and finer scales.

Considering first the SSN, a LP is connected to other LPs at the same resolution with the pattern shown in Figure 3. The influence of the parallel LPs (suggested in [9]) becomes essential in the multi-scale scheme, to avoid duplication of lines caused by the coarse to fine influence.

Considering the choice of the connections between different layers for the grid geometry used (see Figure 4), it's apparent that there is no immediate definition of the LPs above or below a given one.

One possible solution is based on this prescription: if neuron x in scale X is connected to y in scale Y, then conversely y will be connected to x (symmetry in scale). In this case the fine to coarse influence is derived uniquely after defining the coarse to fine one.

Given this, LPs in the coarser scale are connected if they have minimum distance (in the x-y plane) to the given LP. With this definition some LPs will have two LPs above with minimum distance, while others will have one. This asymmetry can be corrected by adjusting the connection weights so that the combined influence of the two minimum distance LPs (defined as weak influence) will be the same as the influence of the single LP in the other case (defined as strong influence), as will be shown in the following section.

2.3 Updating Rule and Look-up Table

Starting from partial "visual" information, the dynamical system of the line and depth processes on the different scales must evolve in time to a state corresponding to a faithful reconstruction of the three dimensional structure and a perceptual grouping of it into "meaningful" pieces. Therefore activation of LPs must be favored either by the presence of a "large" difference in the z values of the nearby DPs or by the presence of a partial discontinuity structure that can be improved. Because usually the perceptual grouping corresponds to the underlying physical structure, these two driving forces cooperate to create the desired results.

Let us define as benefit the square of the derivative at a given point (activation of the LP is "beneficial" when this quantity is large):

Benefit =
$$(\partial z/\partial x)^2 \approx (z_{i+1,j} - z_{i,j})^2/h_k^2$$

for a vertical LP, and let's introduce a cost for a line process in a given environment

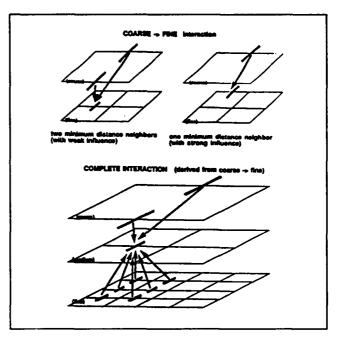


Figure 4: LP neighborhood between different layers.

$$Cost = f(LPs \in SSN; LPs \in DSN)$$

The updating rule for a LP is given by

$$LP \leftarrow 1$$
 iff $Cost < Benefit$

Because the Cost is a positive quantity LPs will be switched on only when there is a difference in nearby z values. Moreover, since the Cost depends on the LPs neighborhood, a good discontinuity structure can be favored by "discounting" Cost if the local structure is improved by the given LP.

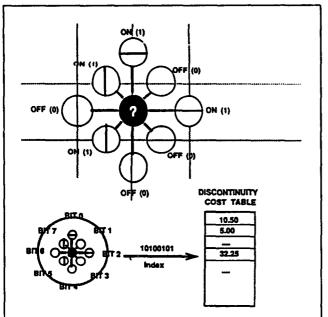
Cost is a function of a limited number of binary variables, therefore to increase simulation speed and to provide a convenient way for simulating different heuristical proposals, a look-up table approach was used.

As shown in Figure 5, values of nearby LPs are used to form an index into the table containing the Cost values ⁴.

2.4 Invariance, Scale and Topology

Segmentation should not depend on the physical scale of the structure. If the depth values of a surface are multiplied by a constant, the same distribution of line

⁴For 6 neighbors one gets a 256 entry table



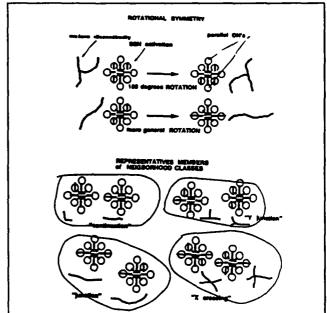


Figure 5: Table look-up for line processes.

processes must be obtained by scaling the Costs appropriately. Besides, the "topological" influence (enforcement of good discontinuity structure) should be independent of scale.

To separate the effects of scale and topology we decided to isolate the scale factor into one parameter dh, corresponding to the typical size of $\partial z/\partial x$ and $\partial z/\partial y$ that we want to be detected by our LPs:

$$Cost_0 \equiv f(0,...,0;0,...,0) = dh^2$$

It would be of little practical use to allow 256 degrees of freedom in the definition of Costs for the SSN. Rotational invariance must be valid. If a given configuration is rotated, Cost must remain equal.

We decided to classify all possible SSN* configurations (let's neglect the effect of the parallel LPs for the moment) into groups, depending on the number of regions in which the surface is divided at the location of the discontinuity. For some examples, see Figure 6. The Cost for a neighborhood with n cuts is multiplied by a parameter a_n . If the number of cuts is too large Cost is set to a very large value (to penalize formation of "tangled" lines).

$$Cost_n \equiv f(LPs \in SSN^*, 0, 0; \underline{0, ..., 0}) = Cost_0 \times a_n$$

if local surface patch is cut into n pieces
$$Cost = \infty \quad \text{if } n \geq 5.$$

Figure 6: Rotational invariance and topological classes.

The "inhibitory" influence of parallel lines is described by factor ai, with

$$Cost(LPs \in SSN; 0,...,0) =$$

$$Cost(LPs \in SSN^*,0,0; 0,...,0) \times a_i^{n||}$$

where $n^{||} \equiv$ number of parallel LPs \in SSN.

Last but not least, presence of lines at the coarser or finer scale will reduce Cost by factors ru or rd respectively, in the strong influence case. In the weak influence case the factors become $\sqrt{r_u}$ or $\sqrt{r_d}$ 5.

$$Cost(LPs \in SSN; LPs \in DSN) =$$

$$Cost(LPs \in SSN; 0,...,0) \times r_u^{n\dagger} \times r_d^{n\dagger}$$

where $n^{\uparrow} \equiv \text{number of above LPs} \in \text{DSN}$

 $(\times 1/2 \text{ if } weak \text{ influence}).$

The parameters a_n were chosen by trial-and-error.

⁵Let's remember that the combined weak influence of two LPs (equal to $\sqrt{r_u} \times \sqrt{r_u}$) must be equal to the strong influence of a single LP (equal to r_u).

3 Parallel Implementation

The multigrid algorithm described in the previous section can be executed in different ways on a parallel computer. One essential distinction that has to be done is related to the number of processors available and the "size" of a single processor.

If implementation is done on a SIMD parallel computer with a number of processors comparable to the number of computational units, the strategy that assigns one processor to each unit (see [4]) obtains the maximum amount of parallelism. The drawback of this approach is that if the implementation is on a hypercube parallel computer and if the mapping is such that all the communications paths in the pyramid are mapped into communication paths in the hypercube with length bounded by two [4], a fraction of the nodes is never used (one third for two-dimensional problems encountered in vision). Furthermore, if the standard multigrid algorithm is used, when iteration is on a coarse scale all the nodes in the other scales (i.e. the majority of nodes) are idle and the efficiency of computation is in part compromised. To ameliorate this problem, intrinsically parallel multiscale algorithms must be considered [5].

Fortunately, for a MIMD computer with powerful processors, sufficient distributed memory and twodimensional internode connections (in particular the hypercube contains a two dimensional mesh), the above problems do not exist.

In this case a two-dimensional domain decomposition can be used efficiently: a slice of the image with its associated pyramidal structure is assigned to each processor. All nodes are working all the time, switching between different levels of th pyramid as illustrated in Figure 7.

No modification to the sequential algorithm is needed for points in the image belonging to the interior of the assigned domain. On the contrary points on the border need to know values of points assigned to a nearby processor. With this purpose the assigned domain is extended to contain points assigned to nearby processors and a communication step before each iteration on a given layer is responsible for updating this strip so that it contains the correct (most recent) values. Only two exchanges are necessary, as shown in Figure 8.

3.1 Communication Overhead and Complexity

Multigrid algorithms are optimal in the sense that they can compute a solution in time proportional to the number of unknowns. Let's suppose that complexity for the standard algorithm is (asymptotically)

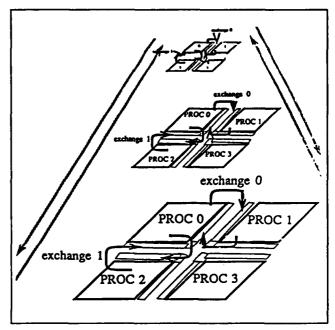


Figure 7: Domain decomposition for multigrid computation. Processor communication is on a two-dimensional grid, each processor operates at all levels of the pyramid.

Time = α n, where n is the number of pixels and α depends on the details of the algorithm.

To a good approximation the complexity for the parallel version is:

$$Time = \alpha \frac{n}{D} + \beta \sqrt{\frac{n}{D}}$$
 (5)

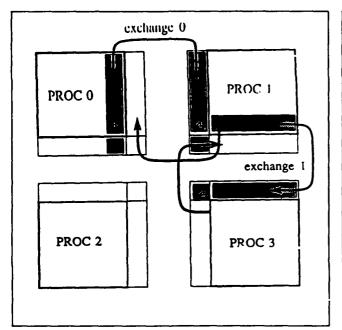
where D is the number of domains (equal to the number of processors). The communication overhead is a "surface effect" proportional to the linear dimension of the domain. The proportionality factor β depends on the number of iterations and on the height of the pyramidal structure.

Preliminary timing has been done using a board with four processors 6 obtaining times of 600-900ms for 65×65 images. Each node spends approximately 20% of its time in internode communication. In addition some time is required to load the data and read results. Results are illustrated graphically in Figure 9.

4 Results: Shape from Shading

An iterative scheme for solving the shape form shading problem has been proposed in [8]. A preliminary phase

⁶Definicom board with Transputers, software from Parasoft



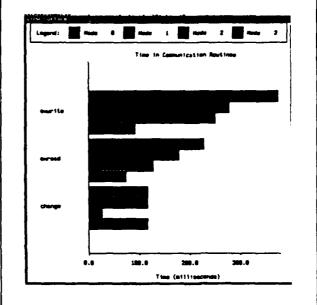


Figure 8: Communication strategy: each node contains a strip of data assigned to nearby processors. Values are updated before each iteration using exchanges in the two directions.

Figure 9: Timing results. Above: time spent exchanging data (change) and communicating with the host (read-write).

recovers information about orientation of the planes tangent to the surface at each point by minimizing a functional containing the image irradiance equation and an integrability constraint, as follows:

$$E(p,q) = \int_{Image} (I(x,y) - R(p,q))^2 + \lambda (p_y - q_x)^2 dxdy$$
(6)

where $p = \partial z/\partial x$, $q = \partial z/\partial y$, I measured intensity, and R theoretical reflectance function.

After the tangent planes are available, the surface z is reconstructed minimizing the following functional:

$$E(z) = \int_{Image} (z_x - p)^2 + (z_y - q)^2 dx dy \qquad (7)$$

Figure 10 shows the reconstruction of the shape of an hemispherical surface starting from a ray-traced image ⁷. Above is the result of standard relaxation after 100 sweeps, below the "minimal multigrid" result ⁸ whose total solution time is equivalent to approximately four iterations on the finest grid.

This case is particularly hard for a standard relaxation approach. The image can be interpreted "legally" in two possible ways: either as a concave or a convex hemisphere. Starting from random initial values, some image patches will "vote" for one or the other interpretation and try to extend the local interpretation to a global one. This not only takes time (given the local nature of the updating rule) but encounters an endless struggle in the regions that mark the border between different interpretations. The multigrid approach solves this "democratic impasse" on the coarsest grids (much faster because now information spreads over large distances) and propagates this decision to the finer grids, that will now concentrate their efforts on refining the initial approximation.

Another example is shown in Figure 11, where the three dimensional structure of the Mona Lisa face painted by Leonardo 9 is reconstructed.

⁷A simple Lambertian reflection model is used.

⁸V cycles with one relaxation on each level

⁹Anticipating the reader's unhappiness with her aesthetic appearance, let's remember that the Lambertian reflectance model is a very naive approximation of the artistic shading used by Leonardo

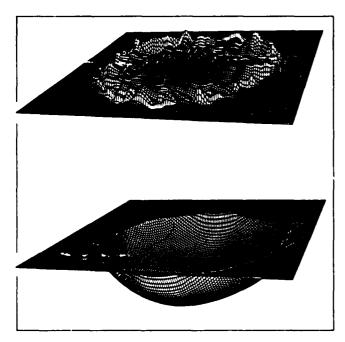


Figure 10: Reconstruction of shape from shading: standard relaxation versus multigrid.

5 Results: Surface Reconstruction from Depth Constraints

The functional for the surface reconstruction problem is:

$$E(z(x,y)) = \int_{Image} (z(x,y) - d(x,y))^2 + \lambda(z_x^2 + z_y^2) dxdy$$
(8)

A physical analogy is that of fitting the data !(x, y) with a membrane pulled by springs connected to them. A given z value is updated as follows:

$$z(x,y) \leftarrow \frac{z_{sum} + \beta h^2 d(x,y)}{n_{sum} + \beta h^2}$$
 (9)

where $h \equiv \text{grid step.}$

$$z_{sum} = \sum_{dx=\pm h; dy=\pm h} \overline{LP}(x+dx, y+dy) \ z(x+dx, y+dy);$$

$$n_{sum} = \sum_{dx=\pm h, dy=\pm h} \overline{LP}(x+dx, y+dy);$$

The effect of active discontinuities (LP=1) is that of inhibiting the smoothing action at their location.

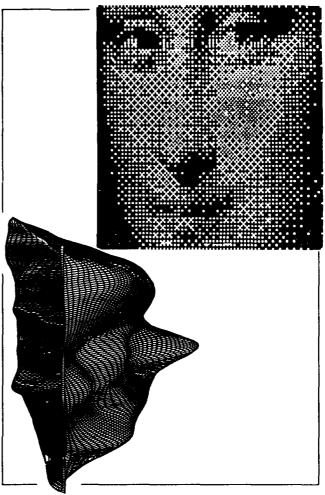


Figure 11: Mona Lisa in three dimensions.

Detailed performance tests have been made using noisy data for z values corresponding to "Randomville" structures. These are obtained by generating random coordinates, heights, slants and tilts for quadrangular blocks and placing 'hem in the image plane. The data are then corrupted by noise and loaded as constraints in the algorithm.

For 129×129 "images" and noise values corresponding to 25% of the highest structure, a faithful reconstruction of the surface (within a few percent of the original one) is normally obtained after one single multiscale sweep (with V cycles) on four layers ¹⁰.

The total computational time corresponds approximately to the time required by 3 relaxations on the finest grid. Because of the optimality of multiscale methods, time increases linearly with the number of image pixels.

¹⁰ In other words, parameters na, nb, nc in mg() are equal to one.

User interfaces examples and results from some tests are shown in the last figures. Figure 12 shows the simulation environment on the SUN workstation, Figure 13 and Figure 14 show the reconstruction of a typical "Randomville" image. The original surface, the surface corrupted by noise (25 %) and reconstruction on different scales are shown in this order.

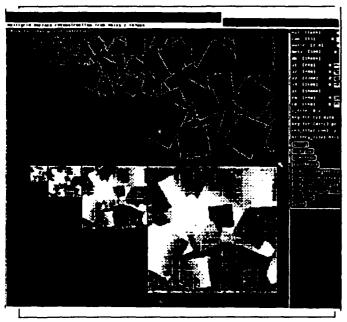


Figure 12: Simulation environment.

6 Summary and Discussion

The extension of multiscale methods to encorporate discontinuity detection can be done in an effective way, combining reconstruction and discontinuity detection in time and scale.

This reduces total computational time by orders of magnitude with respect to single scale methods and provides a better coordination between the two requirements of faithful reconstruction and good discontinuity structure.

The algorithm can be efficiently executed on a parallel computer and a two-dimensional domain decomposition is an effective approach.

Acknowledgment

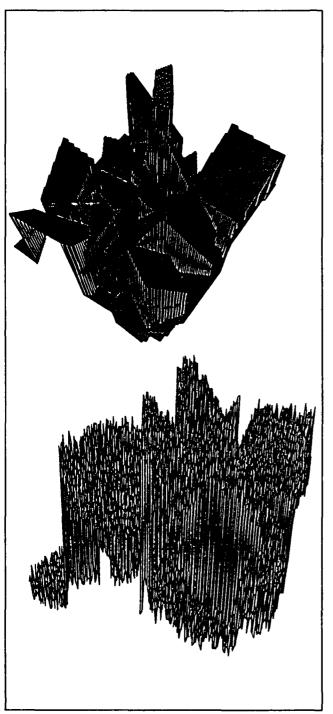
This work was completed while the author was at the California Institute of Technology and benefited from the advice of Dr. Geoffrey Fox and form useful discussions with Christof Koch, Wojtek Furmanski, Paul

Messina, Edoardo Amaldi, Roy Williams and last but not least the people of Parasoft Inc.

Work supported in part by DOE grant DE-FG-03-85ER25009, the Program Manager of the Joint Tactical Fusion Office, the National Science Foundation with grant IST-8700064 and by IBM.

References

- Battiti, R. (1988) Collective Stereopsis on the Hypercube.
 In Proceedings of the III Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, Vol. II, pp. 1000-1006.
- [2] Battiti, R. (1988) Surface Reconstruction and Discontinuity Detection: a Hierarchical Approach. Caltech C3P Report 676 B.
- [3] Brandt, A. (1977) Multi-level adaptive solutions to boundary-value problems, Math. Comput., 31, pp. 333-390.
- [4] Chan, T.F. & Saad, Y. (1986) Multigrid Algorithms on the Hypercube Multiprocessor, IEEE Trans. on Computers, Vol. C-35, No. 11.
- [5] Frederickson, P. & McBryan, O. A. (1988) Intrinsically Parallel Multiscale Algorithms for Hypercubes. In Proceedings of the III Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, Vol II, pp. 1726-1734.
- [6] Furmanski, W. & Fox, G. C. (1988) Integrated vision project on the computer network. Caltech C3P report 623.
- [7] Geman, S. & Geman, D. (1984) Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images, IEEE Trans. Pattern Analysis Machine Intelligence, 6, pp. 721.
- [8] Horn, B.K.P. & Brooks, M.J. (1985) The Variational Approach to Shape from Shading, MIT A.I. Memo 813.
- [9] Koch, C., Marroquin, J. & Yuille A. (1986) Analog neuronal networks in early vision, Proceedins Natl. Acad. Science USA, 83, pp. 4263-4267.
- [10] Marr, D. & Poggio, T. (1976) Cooperative computation of stereo disparity, Science, 195, pp. 283-287.
- [11] Marroquin, J.L. (1984) Surface Reconstruction Preserving Discontinuities, MIT A.I. Memo 792.
- [12] Poggio ,T., Torre ,V. & Koch,C. (1985) Computational vision and regularization theory, Nature, 317, pp. 314-319.
- [13] Stüben, K. & Trottenberg (1982) Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications. In Multigrid Methods Proc., Springer-Verlag, Berlin, pp. 1-176.
- [14] Terzopoulos, D. (1986) Image analysis using multigrid relaxation methods, IEEE Transactions Pattern Analysis Machine Intelligence, 8, pp. 129-139.
- [15] Terzopoulos, D. (1986) Regularization of inverse visual problems involving discontinuities, IEEE Transactions Pattern Analysis Machine Intelligence, 8, pp. 413.



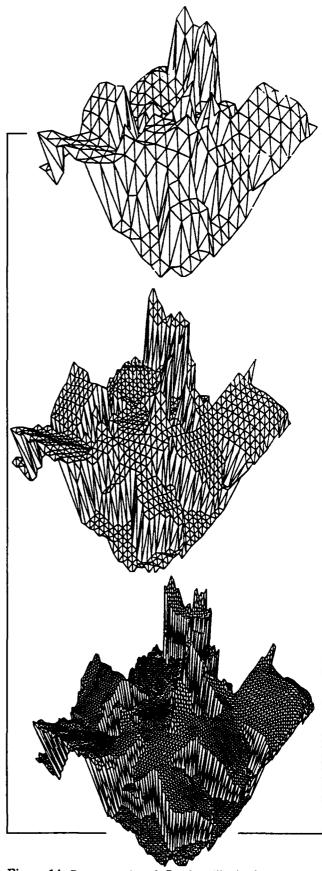


Figure 13: Reconstruction of "Randomville" landscape: origi- Figure 14: Reconstruction of "Randomville" landscape: results nal and noisy images.

on different scales.

An Adaptive Multiscale Scheme for Real-Time Motion Field Estimation

Roberto Battiti
105 Via Menguzzato, 38100 Trento, Italy
E-mail: battiti%genova.infn.it@iboinfn.bitnet

Abstract

The problem considered in this work is that of estimating the motion field (i.e. the projection of the velocity field onto the image plane) from a temporal sequence of images.

Generic images contain different objects with diverse spatial frequencies and motion amplitudes. To deal with this complex environment in a fast and effective way, biological visual systems use parallel processing, visual channels at different resolutions and adaptive mechanisms. In this paper a new adaptive multiscale scheme is proposed, in which the spatial discretization scale is based on a local estimate of the errors involved. Considering the constraints for real-time operation, flexibility and portability, the scheme can be implemented on MIMD parallel computers with medium size grains with high efficiency.

Tests with ray-traced and video-acquired images for different motion ranges show that this method produces a better estimation with respect to the homogeneous (non-adaptive) multiscale method.

1 Introduction

Many low- and medium-level computer vision problems can be formulated in the context of partial differential equations. These in turn are transformed into large algebraic systems (after discretization) that can be solved using iterative relaxation methods. Homogeneous multiscale techniques have been proposed as a way to accelerate the convergence. Finally parallel computation is a natural way to further reduce the solution time in order to obtain real-time or close-to-real-time performance.

While this framework is now widely popular in computer vision, the focus of this work is on an adaptive modification of the previous scheme. Adaptive discretization is not introduced to reduce the computational burden with respect to the homogeneous multiscale strategy but to produce a more reliable estimation of the motion field. The adaptive solution grid is necessary in order to deal with the errors introduced in the definition (not in the solution) of the algebraic system.

In standard reference texts on multiscale and multigrid methods (see for example [4]) one assumes that the coefficients and the inhomogeneous term in the p.d.e. are known precisely (at the finest scale). The real world situation in computer vision is very different. Considering the estimation of the motion field, the p.d.e. coefficients depend on temporal and spatial derivatives of the image brightness pattern. Unfortunately errors are introduced in many ways. First the acquisition process produces quantization errors (due to the finite number of gray values) and possibly random noise. In addition the discretized derivative estimation formulas are valid when the step size can be considered small with respect to the dominant wavelengths contained in the Fourier transform of the image and with respect to the movements in the scene. In general, evaluation at coarse spatial scale will suffer from quantization noise (because the spatial derivatives will be small), while evaluation at finer scale will tend to be unreliable if short wavelengths are present. The appropriate scale for the definition of the p.d.e. coefficients therefore depends on the estimation errors involved and is in general different for the different parts of the image.

2 Reliable Computation of the Motion Field

In particular situations the apparent motion of the brightness pattern, known as the optical flow, provides a sufficiently accurate estimate of the motion field. Although the adaptive scheme proposed in this paper is applicable to different methods, the discussion will be based on the scheme proposed by Horn and Schunck [12]. They use the assumptions that the image brightness of a given point remains constant over time, and that the optical flow varies smoothly almost everywhere. Satisfaction of these two constraints is formulated as the problem of minimizing a quadratic energy functional (see also [17]). The appropriate Euler-Lagrange equations are then discretized on a single or multiple grids and solved using for example the Gauss-Seidel "relaxation" method. The reader interested in

the detailed derivation is referred to [12,21]. The resulting system of equations (two for every pixel in the image) is:

$$(I_x u_{i,j} + I_y v_{i,j} + I_t)I_x = \frac{\alpha^2}{\Lambda \pi^2} (\bar{u}_{i,j} - u_{i,j})$$
 (1)

$$(I_x u_{i,j} + I_y v_{i,j} + I_t)I_y = \frac{\alpha^2}{\Delta x^2} (\bar{v}_{i,j} - 4v_{i,j})$$
 (2)

where $u_{i,j} = \frac{dx}{dt}$ and $v_{i,j} = \frac{dy}{dt}$ are the optical flow variables to be determined, I_x , I_y , I_t are the partial derivatives of the image brightness with respect to space and time, \bar{u} and \bar{v} are local averages $(\bar{u}_{i,j} = \frac{1}{4}(u_{i+1,j} + u_{i,j+1} + u_{i-1,j} + u_{i,j-1}))$, Δx is the spatial discretization step, and α controls the smoothness of the estimated optical flow.

Now, the partial derivatives in equations 1 and 2 need to be estimated with discretized formulas starting from brightness values that are quantized (say integers from 0 to n) and noisy. It is easy to show that, given these derivative estimation problems, the optimal step for the discretization grid depends on local properties of the image. Use of a single discretization step produces large errors on some images. Use of a homogeneous multiscale approach where a set of grids at different resolutions is used, may in some cases produce a good estimation on an intermediate grid and a bad one on the final and finest grid. Enkelmann and Glazer [8,11] encountered similar problems.

We propose a method for "tuning" the discretization grid to a measure of the reliability of the optical flow derived at a given scale. This measure will be based on a local estimate of the errors due to noise and discretization.

The rest of this paper is organised as follows. First we discuss some fundamental shortcomings of the homogeneous multiscale version and derive a formula for the error in derivative estimation. Next, we describe our scheme with adaptive discretization and discuss the multiprocessor implementation. Finally, we present some experimental results obtained with some image sequences.

3 Errors in Derivative Estimation

The difficulties introduced by erroneous derivative estimation can be illustrated with the following one-dimensional example. Let's suppose that the intensity pattern observed is a superposition of two sine waves of different wavelengths:

$$I(x,t) \propto (1+R+\sin(\frac{2\pi}{6}(x-2t))+R\sin(\frac{2\pi}{3}(x-2t)))$$
(3)

where R is the ratio of short to long wavelength components. Using the *brightness constancy* assumption [12] the measured velocity \tilde{v} is given by:

$$\tilde{v} = -\tilde{I}_t/\tilde{I}_x \tag{4}$$

where \tilde{I}_x and \tilde{I}_t are the three-point approximations of the spatial and temporal brightness derivatives¹.

Now, if one calculates the estimated velocity on two different grids, with spatial step Δx equal to 1 and 2, as a function of the parameter R one obtains the result illustrated in Figure. 1. While on the coarser grid the

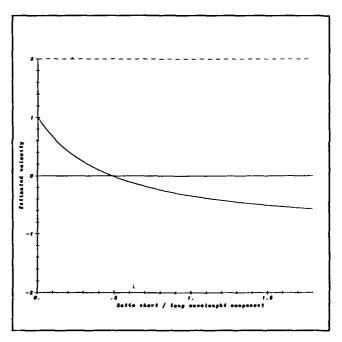


Figure 1: Measured velocity for superposition of sinusoidal patters as a function of the ratio of short to long wavelength components. Dashed line: estimation with $\Delta z = 2$, continuous line: estimation with $\Delta z = 1$, The correct velocity is equal to 2 ($\Delta t = 1$).

correct velocity is obtained (in this case), on the finer one the measured velocity depends on the value of R. In particular, if R is greater than 0.5 a velocity in the opposite direction is obtained!

That is $\tilde{I}_z = \frac{1}{2\Delta z}(I(z + \Delta z) - I(z - \Delta z))$ and analogously for \tilde{I}_z .

For a general one-dimensional profile I(x - vt) it is easy to derive (using Taylor expansion) the following approximation for the relative velocity error due to discretization:

$$\frac{\delta v}{v} = \frac{\tilde{v} - v}{v} \approx \frac{I'''(y)}{6I'(y)} ((v\Delta t)^2 - (\Delta x)^2)$$
 (5)

Considering also the errors introduced by quantization one obtains (after substitutions and reasonable assumptions described in [2]):

$$\mid \frac{\delta v}{v} \mid \approx \frac{C}{\rho^2 n^2} \mid (\Delta_t I)^2 - (\Delta_x I)^2 \mid + \sqrt{\frac{1}{(\Delta_x I)^2} + \frac{1}{(\Delta_t I)^2}}$$
(6)

where $\Delta_x I$ and $\Delta_t I$, are the spatial and temporal differences in intensity values2. These differences grow linearly with the number of discretization levels n. Therefore, while the first term of the overall relative error does not depend on n, the second term, which expresses the contribution due to the quantization process, decreases with n and can be reduced by increasing the number of quantization levels. C is assumed to be a constant (with an heuristic value of $\frac{2\pi}{3}$ derived in the case of sinusoidal patterns). Finally the parameter ρ (fractional range of intensity values in a given image) is needed in the case of over- or under-exposed images. The two-dimensional estimate of the overall relative error is obtained from eqn. 6 by rotational invariance, substituting $(\Delta_x I)^2$ with $(\Delta_x I)^2 + (\Delta_y I)^2$. This amounts to measuring the field unreliability according to the error in the component of the velocity that is normal to the brightness gradient.

4 Adaptive Multiscale Solution on a Multicomputer

The previous example and considerations suggest a new strategy. First a Gaussian pyramid [5] is computed from the given images. This consists of a hierarchy of images obtained filtering the original ones with Gaussian filters of progressively larger size³.

Then the optical flow field is computed at the coarsest scale using relaxation, and the estimated error according to eqn. 6 is calculated for every pixel. If this quantity is less than a given threshold T_{err} , the current value of the flow is interpolated to the finer resolutions

without further processing. This is done by setting an inhibition flag contained in the grid points of the pyramidal structure, so that these points do not participate in the relaxation process. On the contrary, if the error is larger than T_{err}, the approximation is relaxed on a finer scale and the entire process is repeated until the finest scale is reached. A local inhomogeneous approach is thus obtained, where areas of the images characterized by different spatial frequencies or by different motion amplitudes are processed at the appropriate resolutions, avoiding corruption of good estimates by inconsistent information from a different scale (the effect shown in the previous example). The optimal grid structure for a given image is translated into a pattern of active and inhibited grid points in the pyramid, as illustrated in Figure 2.

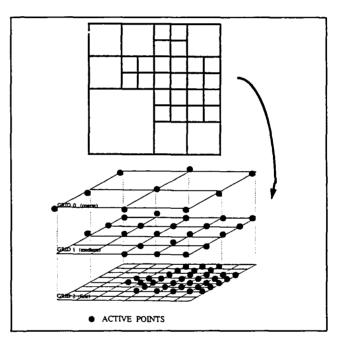


Figure 2: Adaptive grid and activity pattern in the multiresolution pyramid.

The motivation for freezing the motion field as soon as the error is below threshold is that the estimation of the error may itself become incorrect at finer scales and therefore useless in the decision process. It is important to point out that single scale or homogeneous approaches cannot solve adequately the above problem. Intuitively what happens in the adaptive multiscale approach is that the velocity is frozen as soon as the spatial and temporal differences at a given scale are big enough to avoid quantisation errors but small enough to avoid errors in the use of discretised formulas. The only assumption made in this scheme is that the largest

²That is $\Delta_z I = (I(z + \Delta z) - I(z - \Delta z))$ and analogously for $\Delta_z I$

 $[\]Delta_t I$.

Three levels are used for 65x65 images.

motion in the scene can be reliably computed at one of the used resolutions. If the images contain motion discontinuities, line processes (indicating the presence of these discontinuities) are necessary to prevent smoothing where it is not desired (see [1] and the contained references).

The multiscale algorithm described in the previous section can clearly be executed in different ways on a parallel computer (for a pictorial representation see Figure 3).

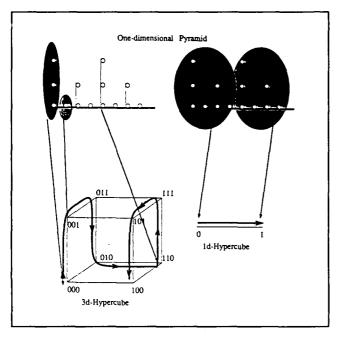


Figure 3: Mapping between a (one-dimensional) multiscale structure and hypercubes of different "grain sizes".

Considering first an implementation on a SIMD parallel computer with a large number of processors, the maximum amount of parallelism is obtained assigning one processor to each grid point [6,13]. Unfortunately this scheme presents a serious disadvantage: while relaxation is executed at a given level of the pyramid all processors assigned to different levels are inactive. In a pyramid with L levels the hardware utilization efficiency will therefore be 1/L. For example, if an image with 512x512 pixels is analyzed at 6 different resolutions the efficiency is at most 0.16.

If the pyramidal structure of grid-points cannot be mapped in a one-to-one manner onto the processing nodes of a given architecture, an additional reduction in hardware utilization efficiency is present. For example, if the implementation is on a fine grain hypercube parallel computer and if the mapping is such that one processor is assigned to each grid point, a fraction of

the nodes is left unassigned [6]. This stems from the fact that the total number of grid-points in the pyramid is not necessarily close to a power of two. For the two-dimensional problem considered, if the number of pixels at the finest resolution is 2^n , the total number of grid-points in the complete pyramid is:

$$2^n \left(1 + \frac{1}{4} + (\frac{1}{4})^2 + \dots + (\frac{1}{4})^{\frac{n}{2}}\right) \approx \frac{4}{3} 2^n$$

Because an (n+1) dimensional hypercube (with 2^{n+1} nodes) is needed, only 66% of the nodes will be assigned to grid points.

The efficiency of the parallel implementation of the multiscale algorithm that assigns one processor to each pixel (or to each grid-point at the finest resolution) is furthermore limited by the fact that when relaxation is executed at coarse resolutions many processors are inactive. A detailed calculation [6] shows that the efficiency decreases at the rate of $\frac{1}{\log_2 I}$ as the grid size I increases ⁴. Considering the communication overhead, the optimal mapping (using hierarchical Gray code [6,9]) is such that the distance between neighboring points on the coarser grids is equal to two, causing some delays especially on old architectures with direct communication limited to neighboring processors.

The above discussion provides some motivation for the use of large grain-size multicomputers (see also [10,7] and [20] for a general discussion).

In this case a simple two-dimensional domain decomposition can be used efficiently: a slice of the image with its associated pyramidal structure is assigned to each processor. Implementing the adaptive strategy, more complex schemes with dynamic load balancing are not needed because a real-time scheme is supposed to produce a solution in the given time in the worst possible case, when all grid units are active (this situation may correspond to images with fine details in all regions of the scene). All nodes are working all the time, switching between different levels of the pyramid as illustrated in Figure 4.

No modification of the sequential algorithm is needed for points in the image belonging to the interior of the assigned domain. On the contrary, points on the domain boundary need to know values of points assigned to nearby processors. With this purpose the domain assigned to each processor is extended with an overlap area and a communication step on a given layer is used before each iteration, as described in [9,15,1]. Only two exchanges are necessary (one in the north-south, and

⁴This result is therefore similar to that obtained in the scheme that assigns one node to each grid point, because $\log_2 I \approx L$, the number of levels.

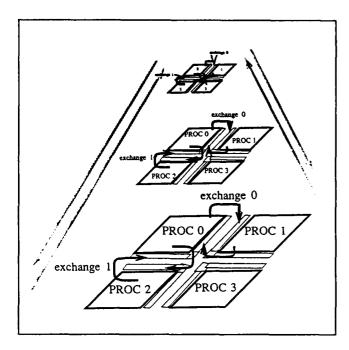


Figure 4: Domain decomposition for multiscale computation. Processor communication is on a two-dimensional grid, each processor operates at all levels of the pyramid.

the other in the east-west direction), as it is shown in Figure 4.

As it will be shown in the examples, the use of limited coarsening [19] is allowed for practical problems in computer vision. In this mapping scheme the coarsest level of the pyramid contains a number of pixels such that each processor will contain at least four of them. In this case the programming environment is more convenient, because no exceptions are needed in the relaxation and communication routines, and the efficiency is optimal, because no processor is idle at the coarsest grids. Eliminating the coarsest grids in the complete pyramid does not affect the solution time in a significant way on the problems considered.

5 A Model of the Architecture and Algorithm

The efficiency of the parallel implementation of a multiscale algorithm depends on characteristics of the algorithm and on the performance of the hardware of a given multicomputer. The following discussion is limited to multicomputers with a two-dimensional grid of connections. We will distinguish parameters related to the image, multiscale pyramid, processors and communications, algorithm, and mapping, as follows:

IMAGE

I Image dimension at the finest resolution (the number of pixels is $I \times I$), where $I = 2^{i} + 1$.

Imin Image dimension at the coarsest resolution considered in the limited coarsening scheme.

PYRAMID

L Number of levels in the complete pyramid. It easy to derive the relation $L = \log_2(I-1) + 1$. The levels are numbered according to: l = 0 (coarsest), 1,..., L-1 (finest).

 $l_{coarsest}$ Level number corresponding to the coarsest resolution used in a given implementation (equal to $\log_2(I_{min}-1)$). l_{finest} is always L-1.

 n_{levels} Number of levels used, equal to $l_{finest} - l_{coarsest} + 1$.

• PROCESSORS AND COMMUNICATIONS

 t_{calc_f} . Time for one floating point operation.

tcalc_i Time for one integer operation.

tcomm_s Startup time for message transmission.

t_{comm_t} Transfer time for transmitting the status of one pixel with the associated line processes (in the present scheme two line processes are associated to each pixel, for example the ones to the north and to the east).

ALGORITHM

W_{relax} Number of operations per pixel during one relaxation step.

W_{disc} Number of operations per pixel for updating the line processes.

W_{init} Number of operations per pixel for initialization.

a Number of relaxations executed on each level.

MAPPING

P Number of processors in the multicomputer. The use of limited coarsening is allowed provided that the following relation is valid: $I_{min} \geq 2 P^{\frac{1}{2}}$ (each processor must contain at least 4 pixels at the coarsest resolution).

E Number of data exchanges per relaxation step.

T Thickness of the overlap area (see previous section).

Tcalc Total calculation time.

T_{start} Total startup time for communication.

Ttrans Total transmission time for communication.

For the performance analysis we follow the model introduced in [9]. The efficiency, or speedup per node, is defined by:

$$\epsilon = \frac{T_{seq}}{P \, T_{conc}(P)} \tag{7}$$

where T_{seq} and T_{conc} are the solution times on the sequential and parallel computer.

In order to calculate the computation and communication times, it is useful to introduce the function $WU_{ms}(\alpha, n_{levels}, \gamma)$, measuring the number of work units required by a given multiscale scheme, where one work unit is defined as the amount of computation required by one relaxation at the finest grid [4]. This function is defined as:

$$WU_{ms}(\alpha, n_{levels}, \gamma) = \alpha + \alpha \sum_{l=1}^{n_{levels}-1} \frac{1}{2^{\gamma l}}$$
(8)
$$= \alpha \frac{1 - 2^{-\gamma n_{levels}}}{1 - 2^{-\gamma}}$$

where γ is the "dimension" of the problem (for example $\gamma=2$ for operations on the two-dimensional image array, $\gamma=1$ for operations on the boundaries).

The complete calculation is composed of an initialization part, where preliminary data are calculated on each layer, and of the multiscale coarse-to-fine scheme. Assuming the use of limited coarsening, the resulting expression for the total computation time is:

$$T_{calc} = \frac{I^2}{P} \left[W_{init} t_{calc_f} W U_{ms} (1, n_{levels}, 2) + (9) \right]$$

$$\left(W_{relax} t_{calc_f} + W_{disc} t_{calc_i} \right) W U_{ms} (\alpha, n_{levels}, 2)$$

The startup time is the same for each level, therefore the total is given by:

$$T_{start} = E t_{comm-s} \alpha n_{levels}$$
 (10)

while the transmission time is

$$T_{trans} = \frac{I}{P^{\frac{1}{2}}} E T t_{comm_t} W U_{ms}(\alpha, n_{levels}, 1) \quad (11)$$

6 Multicomputer Implementation

The hardware parameters for some commercial MIMD computers are collected in Table 1. Data about communication are for message length greater than 1K bytes, t_{comm_byte} is the transfer rate per byte. These data have been collected from different sources [3,18], using different operating systems ⁵ and are not exhaustive about the available multicomputers.

Machine	tcomm_s	tcomm_byte	tcalc_double		
iPSC/1	1000	1.9	41.5 (Int. 80286)		
iPSC/2	660	0.36	6.64 (Int. 80386)		
NCUBE/1	1450	1.68	10.0 (Custom)		
NCUBE/2	?	0.4	0.28 (Custom)		
MEIKO	288	1.19	.8 (Transp.)		
MarkIII fp	136	0.54	0.08 (Weitek)		
Symult S2010	917	0.68	8.3 (Mot. 68020)		

Table 1: Latency, communication time and floating point performance of some multicomputers. Times are in μs .

In the following we present the theoretical speedup that can be expected for machines based on the Transputer. For our problem $W_{init} = 30$; $W_{relax} = 25$; $W_{disc} = 18$. Two relaxations on each level are assumed ($\alpha = 2$). The minimum image size considered for a fixed number of nodes is such that at least four pixels are assigned to each processor. The number of levels used in equations 9-11 is the maximum compatible with limited coarsening, as described in the previous section. In addition, the number of bytes per pixel to transfer is 6, and therefore $t_{comm_t} = 6t_{comm_byte}$. The thickness of the overlap area and the number of exchanges per iteration are T = 1 and E = 2, respectively.

Figure 5 shows the theoretical efficiency calculated for the cited machine as a function of the image size I, using the parameters defined in Table 1. As it can be seen, the efficiency for large images ($I \geq 512$) is greater than 80% if the number of processors is less than 256, approximately. A lower bound⁶ to the total solution time for the different configurations is illustrated in Figure 6. From the graphs it is clear that real-time multiscale processing is within the reach of digital multiprocessor technology. The time for loading the image to the different processors and for obtaining the final results has not been considered and can be a serious limiting factor for two-dimensional architectures

⁵In particular Express from Parasoft has been used on NCUBE/1, Symult S2010, MarkIII and Meiko.

⁶Inefficiencies caused by the use of high-level languages and additional overheads caused by the operating systems are not taken into account.

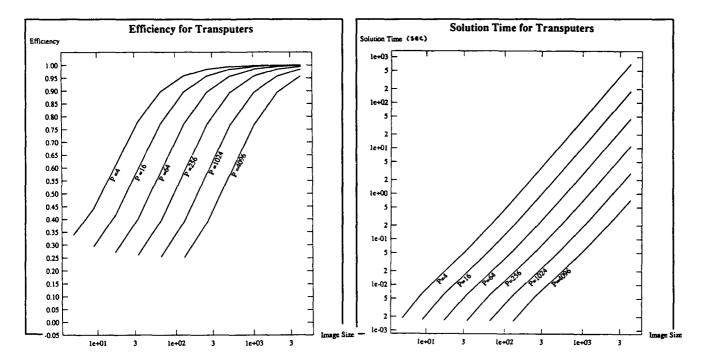


Figure 5: Theoretical speedup for the implementation on the Transputers as a function of the image size. Different curves show the efficiency for different numbers of processing nodes.

(for a review of different image processing architectures see [16,22]). The total processing time can be reduced if overlap of communication and calculation is allowed (see for example [3]). In this case relaxation could be executed first for the grid points on the boundary. Values pertaining to these points could then be exchanged while the interior points are updated.

A program implementing the multiscale scheme has been written in C language and tested on a commercial parallel processor ⁷. Because the communication part is limited to the exchange step repeated before each iteration, the software is easily portable to different multiprocessors. The purpose of this implementation has been that of obtaining some concrete experience and not that of benchmarking different machines. In particular the portability implied by the use of a high-level language and a user-friendly communication environment ⁸ has been obtained at the price of slowing down the theoretical performance by a factor of two-four, depending on implementation details.

Preliminary timing has been done using a board with eight processors ⁹, and test images of 129x129 pixels. The total solution time is of the order of one-three seconds, depending on available memory, compiler and

Figure 6: Total solution time for different configurations.

operating system's version. The obtained efficiency is $\epsilon = 0.87$.

7 Test: Expanding Sphere

These examples show that relaxation per se in the homogeneous scheme does not reduce the solution error in all cases. In some cases too many relaxation steps may increase the error either because of smoothing over regions with rapidly varying velocity fields or because of propagation of constraints referring to different moving objects (if appropriate line processes are not activated).

The first set of images consist of ray-traced expanding spheres superimposed onto a fixed "natural" background ¹⁰. These images contain a unique dominant spatial frequency of the order of magnitude given by the sphere diameter. If we do not consider the effect of quantisation and assume that the motion amplitude is very small with respect to the radius, one iteration is sufficient to recover the correct optical flow (this is the special case of a velocity vector parallel to the brightness gradient). The function of relaxation is, in this case, to provide a better estimate by averaging noisy derivative estimations on neighborhoods with a size that increases with the number of relaxations applied.

⁷A Transputer board hosted by a Sun workstation.

⁸Provided by the Express software package from Parasoft.

⁹Definicom board with Transputers, software from Parasoft

¹⁰ The background is used in order to obtain non-zero derivatives in this region. In fact, if they vanish, all motion fields minimise the Horn and Schunck functional.

Unfortunately, this is true only if one assumes that the occluding boundary is known a priori and if the correct velocity is given on this boundary, as it is the case in Tersopoulos' work [21]. In the general case (no initial information) different results are possible. As will be shown in the following tests, the r.m.s. error increases for small spheres (because noisy information on the boundary is propagated in both directions), while for larger spheres it first decreases (for the averaging effect) but after a few iterations increases (an average over very large neighborhoods becomes worse than the original estimate) with a speed proportional to the parameter α in the cited equations.

The graphs in Figure 7 show the behavior of the r.m.s. error as a function of work units, for two different values of the sphere radius (55 and 95 pixels). Movement is an expansion (3 pixels per frame on the border of the sphere). Both the single scale and the multiscale algorithms are tested.

For the smaller sphere, single scale relaxations make the error worse. The multiscale algorithm does not improve the result. The r.m.s. error as a function of work units in not monotonic (see graph), and the last fine scale iterations show an increasing error. The effect of the boundary is present in particular at the coarsest scale because the ratio boundary / internal points is large.

For the larger sphere (the sphere boundary is now outside the visible window of 129x129 pixels) the situation is different. Single scale relaxations improve the r.m.s. error at the beginning. The error reaches its minimum when 4 work units are completed, then it increases. In this case the multiscale approach reduces the error faster (the minimum is reached after 1.06 work units). But the minimum value is reached on the middle scale and error becomes larger on the finest scale, another case in favor of the adaptive approach. The following figure shows the optical field obtained with the multiscale algorithm in the two cases. These examples show that the effect of the boundary conditions on the result is indeed an important one. Going from an exact a priori knowledge of the occluding boundary with their velocity values to a situation where the only boundary conditions are the "free" boundary conditions at the border of the image, leads in general to very different results11.

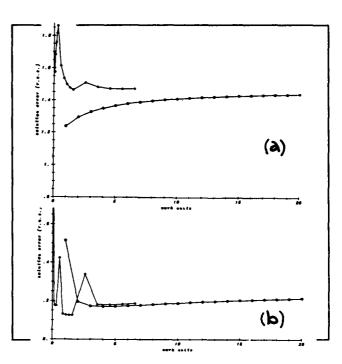


Figure 7: Expanding sphere: r.m.s. error as a function of the amount of computation in the multiscale scheme. Fig.(a): small sphere (radius=55). Fig.(b): larger sphere (radius=95). Single scale results (o) and multiple scale results (□). Interpolation to finer scales increases temporarily the r.m.s. error. Algorithm is terminated after the given number of work units because r.m.s. error is increasing.

If an exact knowledge of the occluding boundary information is missing, incorporation of the boundary detection step described in [1] is essential in order to avoid smoothing across regions corresponding to different moving objects, as it will be shown in the next section.

7.1 Occluding Objects

This test compares the result obtained with or without discontinuity detection. It shows that the optical flow near an occluding boundary may be reconstructed with large errors unless the smoothing process is blocked by line processes.

The images contain two spheres of different sizes (radius are 35 and 24 pixels), translating with velocities (0.0, -1.0) and (0.8, 0.2) against a natural background. Their reflectance patterns are sinusoidal grids (L is 13.3) of a different intensity range mapped onto them using polar projection (in order to obtain a wide range of Fourier components in the different regions of the spheres), while illumination is coming from a source at infinity orthogonal to the image plane. The parameters for the discontinuity detection process are

¹¹Lee et al. [14] show that free boundary conditions make the problem *ill-conditioned*. It is essential to stress that this occurs when the spatial derivatives I_x and I_y are very small, precisely the case that has to be avoided because it is plagued by large errors. If the derivatives are not small, ill-conditioning is not a problem. Dirichlet boundary conditions, that are apparently suggested in the cited reference, assume the precise knowledge of the velocity field on the boundary and this is not the typical case in computer vision.

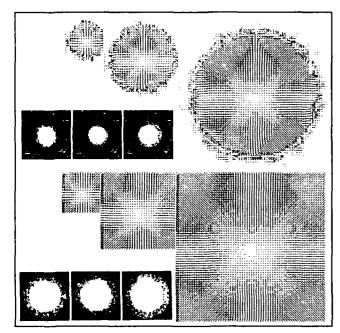


Figure 8: Multiscale optical flow for spheres of different sizes. The effect of the sphere boundary on the result is visible for the smaller sphere.

described in [1].

Figure 9 illustrate the optical flow obtained with the adaptive multiscale process, using 6 relaxations on each of three scales. The first image shows the result obtained without discontinuity detection, while the second one shows the result when a discontinuity detection step has been done every 2 relaxations.

The qualitative results are confirmed considering the r.m.s. error in the optical field for the two cases [2].

8 Test: Natural Image

The images used for this test show a pine cone moving in the upward direction. They were acquired with a S-VHS video camera and a Targa frame grabber. Movement was executed by adjusting a tripod sustaining the object by 0.25 cm every frame. Measured velocity in pixels is 1.6 pixel / frame. Tests have been done for sets of three images taken every one, two, and three frames. The average velocity (on a window centered on the pine cone) obtained with the homogeneous multiscale algorithm is compared with that obtained with the adaptive version. While this second version always produces a better estimate, the difference is particularly significant for large motion amplitudes, as shown in Figure 10.

In this case the fine scale derivative information is

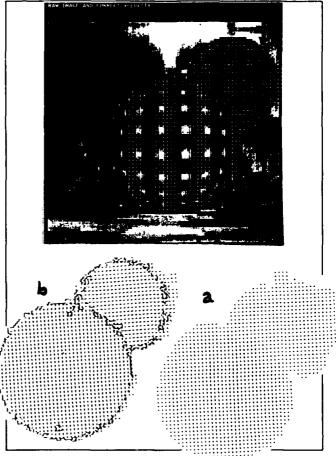


Figure 9: Occluding moving spheres: optical flow obtained without (a), and with the concurrent discontinuity detection process (b).

completely erroneous. This is recognized by the adaptive scheme that freezes the solution obtained at coarser grids.

Acknowledgement

This work was done as a research assistant to Dr. Geoffrey Fox at the California Institute of Technology and benefited in many ways from his suggestions. I thank Christof Koch and Edoardo Amaldi for valuable suggestions.

Work supported in part by DOE grant DE-FG-03-85ER25009, the Program Manager of the Joint Tactical Fusion Office, the National Science Foundation with grant IST-8700064 and by IBM.

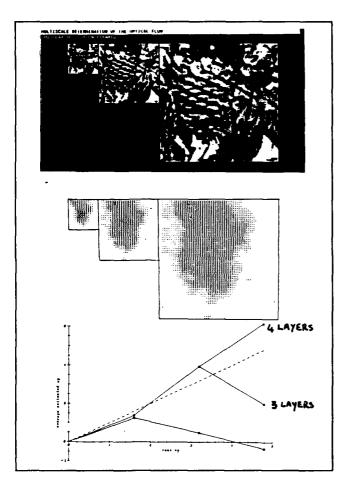
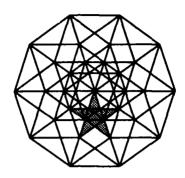


Figure 10: Test images, motion field at different scales, and graph of average velocity. Velocities obtained with the homogeneous (\square) and adaptive (\circ) methods are compared with the correct velocity (dashed line).

References

- Battiti, R. (1989) Surface Reconstruction and Discontinuity Detection: a Fast Hierarchical Approach on a Two-Dimensional Mesh. In Proceedings of the IV Conference on Hypercube Concurrent Computers and Applications, Monterey CA.
- [2] Battiti, R. (1990) Multiscale Methods, Parallel Computation and Neural Networks for Real-Time Computer Vision, Ph.D. Dissertation, California Institute of Technology.
- [3] Bomans, L. & Roose, D. (1989) Benchmarking the iPSC/2 hypercu be multiprocessor, Concurrency: Practice and Experience, 1 (1), pp. 3-18.
- [4] Brandt, A. (1977) Multi-level adaptive solutions to boundary-value problems, Math. Comput., 31, pp. 333-390
- [5] Burt, P.J. (1984) The pyramid as a structure for efficient computation. In Rosenfeld A.(ed.) Multiresolution image processing and analysis, Springer-Verlag, pp. 6-35.

- [6] Chan, T.F. & Saad, Y. (1986) Multigrid Algorithms on the Hypercube Multiprocessor, IEEE Trans. on Computers, Vol. C-35, No. 11.
- [7] Embrechts, H. & Roose, D. (1989) Efficiency and Load Balancing Issues for a Parallel Component Labeling Algorithm. In Proceedings of the IV Conference on Hypercube Concurrent Computers and Applications, Monterey.
- [8] Enkelmann, W. (1988) Investigations of multigrid algorithms for the estimation of optical flow fields in image sequences, Computer Vision, Graphics and Image Processing, 43, pp. 150-177.
- [9] Fox, G., Johnson, M., Lyzenga, G.,Otto, S., Salmon, J. & Walker D. (1988) Solving Problems on Concurrent Processors, Prentice Hall, New Jersey.
- [10] Furmanski, W. & Fox, G. C. (1988) Integrated vision project on the computer network Caltech C3P report 623.
- [11] Glazer, F. (1984) Multilevel relaxation in low-level computer vision. In Rosenfeld A.(ed.) Multiresolution image processing and analysis, Springer-Verlag, pp. 312-330.
- [12] Horn, B.K.P. & Schunck, G. (1981) Determining Optical Flow, Artificial Intelligence, 17, pp. 185-203.
- [13] Ibrahim, H.A.H., Kendler, J.R. & Shaw, D.E. (1987) Low-level image analysis tasks on fine-grained tree-structured SIMD machines, Journal of Parallel and Distributed Computing, 4, pp. 546-574.
- [14] Lee, D., Papageorgiou, A. & Wasilkowski, G. W. (1989) Computing Optical Flow. In Proceedings Conference on Visual Motion, Irvine.
- [15] McBryan, O. & Van de Velde, E. (1987) Hypercube Algorithms and Implementations, SIAM Journal of Scientific and Statistical Computing, 8, pp. 227-287.
- [16] Page, I. (ed.), (1988) Parallel Architectures and Computer Vision, Clarendon Press, Oxford.
- [17] Poggio ,T., Torre ,V. & Koch,C. (1985) Computational vision and regularization theory, Nature, 317, pp. 314– 319.
- [18] Salvador, R. (1989) Message passing benchmark for the NCUBE-1,SYMULT, Mark III and MEIKO, Caltech Concurrent Computation Technical Bulletin 19, California Institute of Technology.
- [19] Solchenbach, K. (1988) Grid applications on distributed memory architectures: implementation and evaluation, Parallel Computing, 7, pp. 341-356.
- [20] Stout, Q.F. (1987) Supporting divide-and-conquer algorithms for image processing, Journal of Parallel and Distributed Computing, 4, pp. 95-115.
- [21] Tersopoulos, D. (1986) Image analysis using multigrid relaxation methods, IEEE Transactions Pattern Analysis Machine Intelligence, 8, pp. 129-139.
- [22] Uhr L. (ed.), (1987) Parallel Computer Vision, Academic Press, Boston.



The Fifth Distributed Memory Computing Conference

8: Ray Tracing

Hypercube Algorithm for Radiosity in a Ray Tracing Environment

Shirley A. Hermitage
Department of Computer Science
Augusta College
Augusta, Georgia

Terrance L. Huntsberger Beverly A. Huntsberger Intelligent Systems Laboratory Department of Computer Science University of South Carolina Columbia, South Carolina 29208

Abstract

Two different approaches to realistic image synthesis are Ray tracing and radiosity. Each method falls short in their attempts to model global illumination present in most environments. A more general model includes the specular and diffuse reflection of both these methods but the combination requires prohibitive computation.

The algorithm presented here extends the standard ray tracing algorithm by adding diffuse rays, while eliminating unnecessary radiosity calculations. The data separation used here is based on the heuristic that light rays, as well as shadow rays intersect objects which are nearby more often those that are more distant. Although that heuristic seems sound, its accuracy is being tested by monitoring the results of processing a large number of different scenes of various complexity.

Keywords: Radiosity, Ray tracing, Parallel graphics algorithms

Introduction

Early attempts at producing realistic computer generated images used shading methods to simulate effects such as shadows, specular reflections and transparency. Development of more accurate image synthesis techniques essentially focused on two distinct methodologies, ray tracing and radiosity. Ray tracing is a view dependent approach to image synthesis which can accurately account for the effects of shadowing and reflections from neighboring surfaces. The method assumes that all light is specularly reflected

or transmitted, and in most cases, ignores any diffuse reflection. Radiosity, on the other hand, attempts to account for a phenomenon that ray tracing ignores, the diffuse inter-reflections of light from surfaces in the scene, which may, in fact, provide most of the illumination. Radiosity methods assume that all surfaces are diffuse reflectors and make no allowances for specular reflection.

Recent research has concentrated on different ways of combining these two effects to create an even more accurate model of global illumination in an environment. Some combination methods add rays of diffusely reflected light to a ray tracing program, while reducing the number of additional rays. Kayija describes a method[8] by which the number of rays that need to be traced can be reduced by stochastic sampling of the rays in the most important directions. Ward et al[13] use a Monte Carlo technique to generate contribution values of indirect illuminance at certain strategic points, and the values are averaged to provide values at other points.

An alternative to including diffuse reflection in a ray tracing program is to add a specular component to the radiosity method. This approach was tried by Immel[7] using a very large system of equations. Wallace[11] used a two-pass approach including a first pass to compute the diffuse component and a second pass which is essentially ray tracing. The weakness of all of these methods is the still the prohibitive amount of computation. In addition, in the approach suggested by Immel[7] there is a strong dependence between the total number of equations that need to be solved and the specular reflectance of the surfaces in the scene. This dependence eliminates the essential strength of the radiosity approach, independence of the number of equations from the surface characteristics.

The algorithm presented here extends the standard ray tracing algorithm by adding diffuse rays, while eliminating unnecessary radiosity calculations. The technique is based on results obtained by Cohen et al[3] who adapted the radiosity method to provide for faster image generation in an animation environment. Cohen's results suggest that light emitting and primary reflectors probably produce most of the diffuse illumination in a scene. It is possible to differentiate between specularly and diffusely reflected rays by adjusting the level of recursion of a diffusely reflected ray so that it will not continue to be propagated. These results suggest a way of adding a diffuse component to a ray tracing program without making it intractable. The unavoidable increase in computation can be handled effectively by the use of parallel processing.

Ray Tracing

Ray tracing programs for the hypercube architecture were previously implemented successfully by Salmon and Goldsmith[9] and Benner[2]. One of the most time consuming tasks of a ray tracing program is the computation of the points at which a ray intersects objects in the scene. One technique that is used to avoid unnecessary intersection calculations is to surround each object in the scene with a tightly fitting, geometrically simple volume. Kay and Kayija[8] suggested enclosing sets of bounding volumes in still larger bounding volumes so that intersection with large parts of the scene can be ruled out by a simple intersection test with a high level bounding volume.

Radiosity

In most environments there may also be illumination present that cannot be directly defined as having originated from a point source, as is usually the assumption made for ray tracing analysis of a scene. Radiosity techniques attempt to determine precisely how diffuse surfaces act as indirect light sources. This process is based on methods used in thermal engineering to determine the exchange of radiant energy between surfaces. In order to calculate the amount of light energy arriving at a surface, a hypothetical enclosure is constructed consisting of surfaces that completely define the illuminating environment. Inside this enclosure there must be an equilibrium energy balance.

The early work on radiosity assumed that all surfaces of the enclosure were ideal diffuse reflectors, ideal diffuse light emitters or a combination of the two[4]. If

the entire environment, including the enclosure, is subdivided into small enough "patches", then each patch can be considered to be of uniform composition, with uniform illumination, reflection and emission intensities over the surface. The total radiant energy leaving a surface (its radiosity) consists of two components emitted and reflected radiation. The radiosity of one patch can be expressed by the equation:

$$B_i = E_i + p_i H_i , \qquad (1)$$

where:

 B_j = radiosity of patch j: the total rate of radiant energy leaving the surface, in terms of energy per unit time and per unit area. (W/m^2)

 E_j = rate of direct energy emission from patch j per unit time per unit area.

p_j = reflectivity of patch j: fraction of incident light reflected back into enclosure.

 H_j = incident radiant energy arriving at patch j per unit time per unit area.

The reflected light is equal to the light leaving every other surface multiplied by both the fraction of that light which reaches the surface in question, and the reflectivity of the receiving surface. Thus H_j , the incident flux on patch j, is the sum of fluxes from all surfaces in the enclosure that "see" j (it may be that patch j "sees" itself if it is concave). This means that

$$H_j = \sum_{i=1}^{N} B_i F_{ij} , \qquad (2)$$

where:

 B_j = radiosity of patch i. (W/m^2) F_{ij} = form factor: fraction of radiant energy leaving patch i, impinging on patch j.

These equations can be combined to give:

$$B_{j} = E_{j} + p_{j} \sum_{i=1}^{N} B_{i} F_{ij}$$
 (3)

where N is the number of surfaces in the enclosure. Such an equation exists for each of the N patches in the enclosure, yielding a set of N simultaneous equations. Each of the parameters E_j , p_j , and F_j must be known or calculated for each patch. The E_j s are nonzero only at surfaces that provide illumination to the enclosure, such as a diffuse illumination panel, or the first reflection of a directional light source from a diffuse surface. The number of patches does not

depend on viewpoint or resolution and is, therefore, usually less than the number of pixels in the resulting image. However, there are no provisions for a specular reflection component in the calculation, since this component is view dependent.

Ray Tracing with a Diffuse Component

Light that is diffusely reflected into the eye may have arrived at the reflecting surface from any direction. A complete ray tracing solution would require that a ray be traced in each one of these directions and propagated if necessary. However, if the light arriving from a particular direction is not coming from a light source, it must be the result of emission or reflection from another surface. If this surface is to contribute illumination of any significance, it will probably be either a light emitter or a primary reflector. It is unlikely that it will be diffusely reflecting light that it has received from another surface. This means that the rays from the original point of intersection need to be traced only in those directions in which another surface is visible and, for all except the mirror direction, should not be propagated beyond the point of intersection with that surface.

In a regular ray tracing program each ray that is specularly reflected or transmitted is traced recursively until either the level of recursion reaches a predetermined depth, or the contribution of the ray is determined to below a specified minimum. It is possible to differentiate between specularly and diffusely reflected rays by adjusting the level of recursion of a diffusely reflected ray so that it will not continue to be propagated.

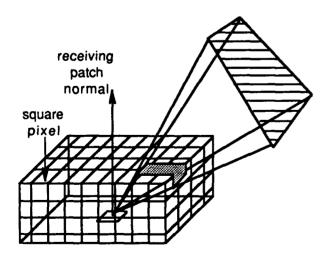


Figure 1. Patch geometry

In radiosity methods a hemi-cube or full-cube is used to determine which other surfaces can be "seen" from a given point. The same approach is used here. The geometry of two patches with the hemi-cube used to determine the fraction of diffusely reflected light reaching each other is shown in Figure 1. This cube will henceforth be referred to as the direction cube. Each cell on the surface of the direction cube represents a direction from which light can reach the point at its center.

Oftentimes, the simplest way to determine whether an object is visible in the direction represented by a particular cell is to project the object onto the face of the cube that contains that particular cell. If the cell is contained in the projection, then the object must be visible in that direction. This is the technique that is used in most radiosity programs, which divide the environment into planar patches that are relatively easy to project. However, if an actual point of intersection with the visible object is also required, a more efficient method for computation of both visibility and intersection point can be found in the various fast ray tracing intersection techniques that have been developed. The fastest algorithm available is that developed by Kay and Kajiya[8] and is the one used in our combined algorithm.

The more accurate lighting model which accounts for both ray tracing and radiosity is expressed as shown in Equation (4). Optimization of the integral calculation is accomplished using the recent results for intersurface projections [1],[10],[12].

$$I_{out}(\Theta_{out}) =$$

$$E(\Theta_{out}) + \int_{\Omega} p''(\Theta_{out}, \Theta_{in}) I_{in}(\Theta_{in}) \cos(\Theta) d\omega \quad (4)$$

where:

 I_{out} = the outgoing intensity for the surface

 I_{in} = an intensity arriving at the surface from

the environment

E = outgoing intensity due to emission by the

surface

 Θ_{out} = outgoing direction

 Θ_{in} = the incoming direction

 Ω = the sphere of incoming directions

θ = the angle between the incoming direction

and the surface normal

 $d\omega$ = the differential solid angle through which

the incoming intensity arrives

p" = the bidirectional reflectance/transmittance

of the surface

The term p'' in the equation is broken into two components for the specular and diffuse components and is given by:

$$p''(\Theta_{out},\Theta_{in}) = k_s p_s(\Theta_{out},\Theta_{in}) + k_d p_d$$

where

= fraction of reflectance that is specular k_d = fraction of reflectance that is diffuse $k_* + k_d = 1.$

Hypercube Implementation of the Model

A basic assumption of the approach to image synthesis (that follows) is that the scene is described in the Haines[5] suggested standard format for a graphics database (nff) and that the program will include code that can read a scene file of this type. This implies that the scene must be composed of a combination of well defined primitives - spheres, cones, cylinders and polygons. Only point light sources are included in the Eric Haines data base. Since light emitting surfaces are to be included, an extra field is added to the object description.

The image is divided into horizontal bands, numbered from 0 to M-1, each of which will be assigned to a different processor. Let a pixel's coordinates be called (I_x, I_y) . Let the number of rows J mapped onto each processor be the number of rows in the image, Xsize, divided by the number of processors, M. To determine which processor I, a given row I_y is mapped onto, a ring-mapping is used so that $I = I_y \operatorname{div} J$. If the scene is projected onto the image using a perspective projection, each pixel band will represent one part of the scene. The view volume for the entire image, is thus subdivided into smaller volumes, one for each band.

Object assignment depends on the dimensions of the object's bounding slabs. A d_{min} and d_{max} in each of the x, y and z directions, is computed for each object as the scene file is read. These define the object's bounding volume which is stored with other object data such as shape and surface characteristics. Intersection of a band's view volume with the bounding slabs of an object determines that the object has an effect on that band. The result is that the object's data is stored on the processor associated with that band. Determination of which processor an object is assigned to, is described below.

= eye location (also center of projection).

R= view reference point.

UP = view-up direction.

d = view distance.

= R - C (normalized).

 $=UP - (N \odot UP)N$

(vertical axis in view plane).

 \boldsymbol{U} $= N \otimes V$

(third axis of viewing coordinate system).

= scalar offset for relative position of I_x and I_y .

1. Project each corner, E, of the object's bounding volume onto the view plane by solving the set of three parametric equations: $E = C + t * d * N + f(I_x) * U + f(I_y) * V$

for the three unknowns: $t, f(I_x), f(I_y)$.

- 2. Obtain the vertical image coordinate of the pixel, onto which this corner projects, from the value of $f(I_y)$, which is a linear expression in I_y .
- 3. Determine the maximum (I_{y_max}) and the minimum (I_{y-min}) of these vertical image coordinate values for all 8 corners of the bounding volume.
- 4. Assign the object to all processors between $I_{y_min}divJ$ and $I_{y_max}divJ$.

Each object in the scene is assigned to one or more pixel bands, and thus to the processor responsible for that pixel band, depending on the dimensions of its bounding slabs. This mapping process is shown in Figure 2.

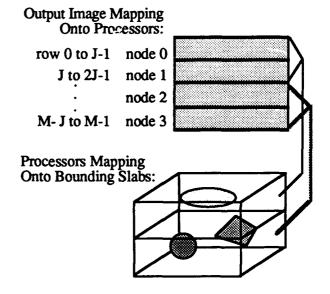


Figure 2. Mapping Process

When more than one object is assigned to a particular processor, the tree of bounding volumes within bounding volumes that was described earlier is used for storage. It is constructed in a manner first suggested by Kay and Kajiya[8] that groups objects by nearness as much as possible.

A ray is traced in its processor until no more intersections with objects occur or a specified maximum depth of recursion is reached. If the maximum is not reached and the ray passes into the space of another processor, then it is added to a list of rays to be passed to that processor. The hypercube ray tracing algorithm with radiosity effects included is given below.

Diffuse Ray Tracing Algorithm

```
Level=0
Weight = 1
Form ray with origin at eye and direction
    through pixel
Intersect ray with everything in this processor I
If intersection occurs
    Find nearest point of intersection P
    Find Normal N to surface at P
    Shade(I, Level, Weight, P, N, raydir, hit, color)
    Mark direction cells at P above or below
    For each cell marked as above surface
       Compute direction, cell<sub>D</sub>
       Create cuberay with:
           origin = P
           direction = cell_D
           weight = N \odot cell_D \times K_d
           level = maxlevel - 1
           label = this pixel
       Trace ( I, cuberay, tcol )
       color = color + tcol \times (N \odot cell_D) \times K_d
    EndFor
Else
    color = bg_{color}
EndIf
```

The Trace algorithm is modified for the hypercube from that described by Heckbert[6]. When a ray is passed to a neighboring processor it takes with it information on ray origin, direction, weight, level and image coordinates of the pixel to which this ray contributes.

Time will be wasted if the processor from which the ray originated suspends processing until it receives a response from the processor to which the ray was passed. If multi-tasking were available, the processor could commence processing the next pixel while awaiting a reply. The was the solution chosen by Salmon and Goldsmith [9] for their hypercube ray tracer, but they had to implement a system of multi-tasking to accomplish this. We used an alternative method in which a list of rays that need more tracing is collected and then passed as a group to a neighboring processor. If tracing is completed in some other processor, that processor needs to know to which pixel this ray contributes color. Rays not traced to their full depth of recursion pass information gathered up to point to the next processor as part of the ray description.

In the modified version of Shade, a color value is initially computed based on the assumption that there are no shadows. A trial shadow ray is then be created and tested first for local shadows by intersecting it with all objects assigned to the current processor. If an intersection is found, the color previously assigned is removed. If there is no intersection with any local object, the shadow ray may be passed to neighboring processors for further testing. Further testing is not necessary when the shadow ray reaches the light source before it leaves the space of the current processor.

Computing the point of intersection of the shadow ray with the plane separating the spaces of two different processors is done as follows: Given that: P = origin of shadow ray, L = direction of shadow ray, and a general point on the line is Q(t) = P + t * L. Q(t) is in the plane separating processor I from processor I-1 if the line joining Q to C (center of projection) lies entirely in the plane and is therefore perpendicular to the plane normal. Therefore: $(P+t*L-C) \odot (U \otimes d*N+K_iV)=0$ can be solved to obtain t at intersection. This value of t is compared to the distance from the shadow ray origin to the light source in order to determine whether to pass the shadow ray to processor I-1.

The shadow ray descriptor that is passed to other processors for testing will have essentially the same format as the ray descriptors passed for tracing, except that its direction is not normalized, and so contains the essential information about the distance of the light from the intersection point. If it is found in subsequent testing that an object assigned to another processor blocks the shadow ray, then the contribution of that light source will be subtracted from the final color value of the pixel.

After a processor completes its own band of pixels, it receives and processes the lists of rays which entered its space from neighboring processors. Depending on the depth of recursion for the diffusely reflected rays, further processing may be required in another node. Finally all the pixels, having been assigned a final value, are merged into a single image file on the host for display.

Discussion

Two different approaches to realistic image synthesis are Ray tracing and radiosity. Each method falls short in their attempts to model the global illumination present in most environments. A more general model includes the specular and diffuse reflection of both these methods but the combination requires prohibitive computation. One way to reduce compute time is to use parallel processing but this alone is not enough. The approach suggested here is to add a diffuse component to each light ray while still exploiting the time savings of the progressive refinement techniques of Cohen et al [3]. In addition, a hypercomputer mapping of the algorithm is presented.

The data separation used here is based on the heuristic that light rays, as well as shadow rays intersect objects which are nearby more often those that are more distant. Although that heuristic seems sound, its accuracy is being tested by monitoring the results of processing a large number of different scenes of various complexity. Another aspect of ray tracing on a distributed system is the issue of load balancing. At each processor, the performance of our combined algorithm depends upon the number of diffusely reflected rays present locally in the scene. Bounding slabs can be used to balance the number of objects in each processor, thus balancing the time needed for the ray tracing. This approach is currently being pursued.

References

- [1] D.R. Baum, H.E. Rushmeier and J.M. Winget. Improving radiosity solutions through the use of analytically determined form-factors. *Computer Graphics, Proc. SIGGRAPH 89*, Vol. 23, No. 3, 1989, pp. 325-334.
- [2] R. Benner. Parallel graphics algorithms on a 1024-processor hypercube. to appear in *Proc. HCCA4*, Monterey, CA, Mar 1989.
- [3] M.F. Cohen, S.E. Chen, J.R. Wallace and D.P. Greenberg. A progressive refinement approach to fast radiosity image generation. Computer Graphics, Proc. SIGGRAPH 88, Vol. 22, No. 4, 1988, pp. 75-84.
- [4] C.M. Goral, K.E. Torrance, D.P. Greenberg and B. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics, Proc.* SIGGRAPH 84, Vol. 18, No. 3, 1984, pp. 213-222.

- [5] E. Haines. A proposal for a standard graphics environment. IEEE Computer Graphics and Applications, Vol. 7, No. 11, Nov 1987, pp. 3-5.
- [6] P.S. Heckbert. Writing a ray tracer. SIGGRAPH 88, Atlanta, GA, Aug 1988, Tutorial notes.
- [7] D.S. Immel, M.F. Cohen and D.P. Greenberg. A radiosity method for non-diffuse environments. Computer Graphics, Proc. SIGGRAPH 86, Vol. 20, No. 4, 1986, pp. 133-142.
- [8] T.L. Kay and J.T. Kajiya. Ray tracing complex scenes. Computer Graphics, Proc. SIGGRAPH 86, Vol. 20, No. 4, 1986, pp. 269-278.
- [9] J. Salmon and J. Goldsmith. A hypercube raytracer. Proc HCCA3, Pasadena, CA, Mar 1988, pp. 1194-1206.
- [10] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. Computer Graphics, Proc. SIGGRAPH 89, Vol. 23, No. 3, 1989, pp. 335-344.
- [11] J.R. Wallace, M.F. Cohen and D.P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. Computer Graphics, Proc. SIGGRAPH 87, Vol. 21, No. 4, 1987, pp. 311-320.
- [12] J.R. Wallace, M.A. Elmquist and E.A. Haines. A ray tracing algorithm for progressive radiosity. Computer Graphics, Proc. SIGGRAPH 89, Vol. 23, No. 3, 1989, pp. 315-324.
- [13] G.J. Ward, F.M. Rubenstein and R.D. Clear. A ray tracing solution. Computer Graphics, Proc. SIGGRAPH 88, Vol. 22, No. 4, 1988, pp. 85-92.

The Hypercube Ray Tracer

Michael B. Carter and Keith A. Teague

Department of Electrical and Computer Engineering
Oklahoma State University

Introduction to Ray Tracing

Ray tracing is presently the method of choice for generating the most realistic looking synthetic images. Ray tracing works by tracing the paths of many rays of light backwards from the viewpoint, through pixels on an imaginary viewplane, and into an environment of mathematically defined 3dimensional solids (objects) called the "scene". These rays are called primary rays. When a primary ray's intersection point with the closest object in the scene is found, a shading model is applied at that point, and the corresponding pixel's brightness (color) is calculated. Often, the shading model will require that secondary rays be traced due to reflection or refraction. Also, the shading model will fire a ray toward each light source in the scene to see if the intersection point is in that light source's shadow. These rays are called shadow rays. Much work has been done by others to optimize various parts of the ray tracing algorithm, such as ray-object intersection. [1, 5, 2]

Scope and goals of research

Very little work has been done, however, in the area of parallel ray tracing. Ray tracing, until now, was largely confined to serial machines. Thus, one purpose of the Hypercube Ray Tracer is to demonstrate that ray tracing is well suited to parallel architectures, and exhibits excellent speedup. Another purpose of the Hypercube ray tracer project is to select and develop algorithms and data structures which lend themselves well to the parallel, distributed computing environment of the iPSC/2.

Parallel issues in ray tracing

A little thought will disclose that the brightness of each pixel on the viewplane is completely independent of its neighbors. This is not to say there is no correlation among pixels, clearly there is, but merely the calculations performed for pixels are independent. This fine grained parallelism makes ray tracing suitable for coarse, medium, and fine grained parallel machines.

Furthermore, since all pixel calculations are independent of one another, there need be no communication between processing elements during

the rendering process itself. This observation makes ray tracing equally attractive to both shared-and distributed memory architectures from an interprocessor communications standpoint. One can envision ray tracing schemes which do perform interprocessor communications for one reason or another. Some of these schemes will be discussed later in this paper.

There is one key data structure that drives the ray tracing algorithm: the database describing the objects to be rendered. This so called "object database" can become very large if the number of objects to be rendered becomes large. Since objects can be reflective and refractive, secondary rays and shadow rays can intersect any object in the scene. For this reason, each processor must have access to the entire object database when rendering a given scene. Though this is not usually a problem for shared memory parallel processors, it can be for distributed memory machines because of the generally limited memory available to individual processors. This issue, too will be discussed later in this paper.

Problem decomposition for the iPSC/2

The ray tracing algorithm itself may be decomposed in many ways onto the hypercube topology of the iPSC/2. One might opt for the simplest approach of placing a complete ray tracer on each node. Each node then ray traces a fixed portion of the pixels on the viewplane. This method has the distinct advantage of simplicity. A little thought will reveal that some portions of an image may take longer to calculate than others. This will be the case for pixels whose primary rays intersect complex, mirrored, or refractive objects. Thus, one node may take much longer to ray trace its fixed portion than other nodes. This leads to poor node utilization and poor load balance.

In a different scheme one might divide the processing nodes into "intersection processors" and "shading processors" with the former doing all ray-object intersections and the latter performing all shading calculations on intersection points found by the intersection processors. [3] Since all iPSC/2 nodes are the same, there is no reason to believe that some nodes could perform one function better than others. Furthermore, the intersection process is far more time consuming than the shading

process, so any time saved in shading calculations would make very little difference in overall performance. This method has the further disadvantage of requiring a considerable amount of internode communications between intersection processors and shading processors. Efficiency would suffer because of this communication time.

The Hypercube Ray Tracer uses a variation on the first alternative presented above; a complete ray tracer is placed on each node. It solves the load balance problem by dividing the image in such a way that all nodes are likely to share in the more difficult portions of the image. We start at the top raster of the image and move down, assigning rasters to successive nodes. When all nodes have been assigned a raster, then assignment restarts with the first node and continues until all rasters have This is called the "comb" been assigned. distribution because the rasters associated with a given node look like the teeth of a comb. Experiments show this image decomposition to produce very low load imbalances in images of medium to large size with an arbitrary number of objects in the scene.

Implementation issues

There are a number of other issues which need attention in the parallel environment of the iPSC/2. These are the structure and content of the object database, the ray-object intersection method, and antialiasing methods. Several choices are available for each topic, and are discussed in turn.

First, we turn our attention to the structure of the object database and ray-object intersection methods. These subjects are closely tied together and should be considered simultaneously. The way in which the database is organized and queried is of critical importance to the speed of ray-object intersection. And since ray-object intersection is the major consumer of time in the ray tracing process, database organization is equally critical in overall rendering speed.

There are two general classes of ray-object intersection acceleration methods, and each dictates a database structure. The two classes are object subdivision [5] and space subdivision [1, 2]. Object subdivision methods usually organize the database into a hierarchy while space subdivision methods divide the database into a number of smaller databases. The object subdivision method of [5] has the advantage of keeping the database in one piece. Since this one-piece structure will aid in later database distribution, we chose it over competing object subdivision techniques.

Each pixel, and thus each node, must have access to the entire object database. This does not mean that each node must have a copy of the database, just access to it. Indeed, if each node

stores a copy of the database, a tremendous aggregate amount of memory will be wasted storing these multiple copies. If, on the other hand, a node must often or frequently request objects from other nodes, then a tremendous amount of time will be wasted waiting for objects to arrive. In the distributed memory environment of the iPSC/2, this tradeoff has serious implications: speed must be traded for larger database sizes. In this first implementation of the Hypercube Ray Tracer, speed of execution and ease of implementation are more important than a large database size. This decision is tempered with one proviso, however. All data structures and algorithms must be suitable for use with the distributed database concept so that future expansion will be simplified as much as possible.

Finally, we shall consider several antialiasing techniques. Aliasing in synthetic images manifests itself in several forms. First, and best known of all aliasing modes, is the "jaggies" -- those jagged edges at the edges of an object's image or shadow. Not so well known are the beat patterns and moire patterns that appear when an area of high spatial frequency is sampled by the ray tracing algorithm. All of these problems are caused by sampling on a regular pixel grid. Antialiasing is very important to realism because it smooths out jagged edges and other artifacts in the image.

Again, there are several methods to choose from. Statistical pixel subsampling techniques are most attractive in terms of ease of implementation. [6] These methods combat aliasing effects by sampling a pixel's area more than once in a nonuniform pattern. Once several samples are taken from the pixel, a statistical test such as a variance threshold is applied to the samples to determine if more samples are needed. When enough samples are accumulated, they are averaged in some way to produce a representative brightness for the pixel in question. Most statistical subsampling techniques require knowlege of brightness levels in only a one pixel neighborhood. This is attractive in the distributed environment of the iPSC/2 because rendered image data is distributed among nodes. Thus, any methods which require the brightness of a neighboring pixel may have to get it from another node -- a relatively slow process on the iPSC/2. Here again, we choose speed and ease of implementation as more important to the Hypercube Ray Tracer.

Other antialiasing methods rely on filtering, adaptive pixel subsampling, or a combination of both. [4, 8] All of these methods present problems because they assume the immediate availability of any pixel's brightness. As stated above, this is not always the case in a distributed computing environment. These methods are rejected for this

reason in this first implementation of the Hypercube Ray Tracer.

One feature of the Hypercube Ray Tracer that has not received a lot of research is the process of constructive solid geometry (CSG). [9] This is the process of defining new object types from previously existing ones by means of boolean operations. For example, a box with a hole in it can be constructed by subtracting a cylinder from the box. (Box and not(Cylinder)) The Hypercube Ray Tracer includes an original ray-CSG object intersection scheme which we have developed on top of Kay's intersection scheme. It performs the intersection in linear time with the number of objects defining the CSG object.

Late in the Hypercube Ray Tracer's construction, a decision had to be made between implementing CSG or distributing the object database. After much thought, we decided that CSG would have a profound impact on the object database organization. It then made sense to implement CSG first. This way, any changes in the database organization could be easily taken into account when the time came to distribute it.

Results

All test images were calculated at 512 x 512 resolution on 32 scalar-enhanced nodes with no antialiasing. Figures 1 and 2 show the amount of time taken to render each test image on different numbers of nodes. Tables 1 and 2 show the same data as well as speedup and efficiency for the same configurations. Here, efficiency is defined as the speedup over the number of nodes used for a given problem.

The Hypercube Ray Tracer's antialiasing strategy greatly increases the visual realism of its rendered images. Jagged edges and razor sharp lines are no longer a problem. A rather substantial time penalty is paid for antialiasing, though. The final method used casts eight uniformly random rays through each pixel and calculates the variance of the average intensity. If the variance is above a predetermined threshold, groups of four additional rays are traced until the variance drops below the threshold or until 32 rays have been traced, whichever comes first. Thus, each pixel is 8 to 32 times more expensive to compute than without antialiasing.

Future Work

Many tradeoffs have been made in the design of the Hypercube Ray Tracer. The most notable of these is the choice to maintain a complete copy of the object database on each node of the iPSC/2. This one decision greatly sped program development but severely limited the maximum number of objects. New techniques, which are already under development, will allow the database to be distributed across the nodes with only a modest performance penalty.

The rather recent development of "distributed ray tracing" [6] adds new and exciting effects to ray traced images such as penumbrae, distributed light sources, and frosted glass. Note that distributed ray tracing should not be confused with the object database distribution proposed above. Rather, it refers to the small random perturbations of rays used to achieve the aforementioned effects. Distributed ray tracing, too, is well suited to the distributed computing environment for the same reason simple ray tracing is: the ray calculations are independent of one another.

Relatively simple object models are used in the Hypercube Ray Tracer. They include spheres, cylinders, cubes, polygonal prisms, and convex superquadric ellipsoids. New shapes such as bicubic patches, superquadric toroids, and swept cubic curves would greatly add to the usability of the Hypercube Ray Tracer.

Summary

Ray tracing is a complex rendering technique which has, until now, been almost exclusively confined to serial computers. The Hypercube Ray Tracer efficiently takes the rendering technique into the parallel domain with considerable time savings and very nearly linear speedup. Ray tracing is thus shown to be well suited to distributed memory parallel architectures.

Leading object intersection algorithms and data structures are chosen and modified to be efficient and expandable in the parallel environment. Antialiasing techniques are discussed and applied in parallel to the problem. The ramifications of object database duplication are discussed at length and considerations are made for future distribution across the iPSC/2.

The problem of load balancing is discussed, and a static load balancing scheme based on the "comb" image decomposition is offered as a primary solution. Results are presented to confirm that the "comb" decomposition is a very effective one with the current repertoire of object database organization and object intersection algorithms.

References

- [1] Arvo, James, and David Kirk (1987) Fast Ray Tracing by Ray Classification, Computer Graphics, 21(4), July 1987, pp. 55-64.
- [2] Fujimoto, Akira, T. Tanaka, and K. Iwata (1986) ARTS: Accelerated Ray Tracing System, *IEEE Computer Graphics and Applications*, 6(4), April 1986, pp. 16-26.

- [3] Gaudet, Severin, Richard Hobson, Pradeep Chilka, and Thomas Calvert (1988) Multiprocessor Experiments for High-Speed Ray Tracing, ACM Transactions on Graphics, 7(3), July 1988, pp. 151-179.
- [4] Heckbert, Paul S. (1986) Survey of Texture Mapping, *IEEE Computer Graphics and* Applications, 6(11), November 1986, pp. 56-67.
- [5] Kay, Timothy L., and James T. Kajiya (1986) Ray Tracing Complex Scenes, ACM SIGGRAPH 1986, 20(4), August 1986, pp. 269-278.
- [6] Lee, Mark, Richard A. Redner, Samuel P. Uselton (1985) Statistically Optimized Sampling for Distributed Ray Tracing, Computer Graphics, 19(3), July 1985, pp. 61-67.
- [7] Lee, Mark, Personal Communication.
- [8] Michell, Don P. (1986) Generating Antialiased Images at Low Sampling Densities, Computer Graphics, 21(4), pp. 65-72.
- [9] Youseef, Saul (1986) A New Algorithm for Object Oriented Ray Tracing, Computer Vision, Graphics, and Image Processing, 34, 1986, pp. 125-137.

Table	1:	Performance	Data	for	Self Portra	ut Image
(1381 phiects)						

 Nodes	Total Time	Node Time	Speedup	Efficiency	Load Imbalance (%)
 1	13834	13834	1.0	-	-
2	13884	6900	2.0	100	.14
4	13912	3453	4.0	100	.32
8	14011	1728	0.8	100	.81
16	14806	867	16.0	100	1.73
32	15038	435	31.8	99.4	3.68

Table 2: Performance Data for Superquadric Grid Image (93 objects)

Nodes	Total Time	Node Time	Speedup	Efficiency	Load Imbalance (%)
1	11468	11468	1.0	•	•
2	11462	5736	2.0	100	.19
4	11463	2869	4.0	100	.42
8	11465	1443	8.0	100	1.32
16	11468	725	15.8	98.7	2.21
32	11468	373	30.7	95.9	8.58

Time vs. Number of Nodes for Self Portrait Image (1381 objects)

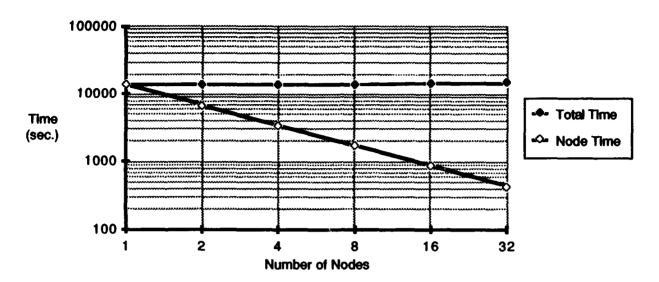
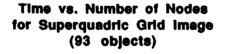


Figure 1: Time vs. Number of Nodes for Self Portrait Image



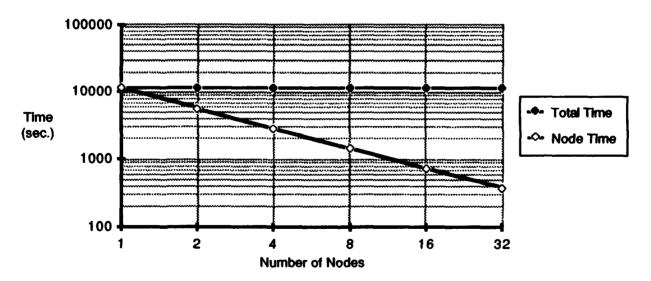


Figure 2: Time vs. Number of Nodes for Superquadric Image

Distributed Object Database Ray Tracing on the Intel iPSC/2 Hypercube

Michael B. Carter and Keith A. Teague

Department of Electrical and Computer Engineering Oklahoma State University 202 Engineering South, Stillwater, OK 74078

Abstract

A medium-grained, distributed-memory parallel computer is used as a platform from which to research a specific issue in accelerating the ray-tracing process. The work presented here deals specifically with the problem of distributing large object databases over the computing nodes of the Intel iPSC/2. An efficient object database decomposition method is presented which behaves like a fully-associative cacne. Ramifications of the distributed object database forced the development of an interruptible ray-tracing loop, a ray scheduler, and a dynamic viewplane decomposition. Performance metrics, such as object database hit ratio, load-balance, and efficiency are presented.

Introduction

Ray-tracing is a realistic image synthesis technique which produces superior quality images by consuming superior amounts of computer time! Although much work has been done to speed the ray-tracing process, it still remains one of the most expensive image synthesis techniques in terms of computer time used [1, 3, 4, 5, 8, 9].

The ray-tracing process, is deceptively simple. Parallelizing the ray-tracing process on a distributed-memory parallel computer also seems simple at first glance. It has, however, several pitfalls which are well hidden. One such pitfall is the problem of distributing the object database (ODB) across the nodes of the computer without seriously affecting the performance of the already power-hungry algorithm. Both the object database decomposition and the problems associated with it will be discussed in subsequent sections.

Before launching into this, however, let us first briefly review the ray tracing algorithm. The ray-tracing milieu consists of an observer, a viewplane, and a set of objects called the scene. The observer is a point in space from whose perspective the scene is to be rendered. The viewplane is an imaginary rectangle through which the observer views the scene. The viewplane is divided into a grid of pixels. It is the task of the ray-tracing procedure to find the light intensity present at each pixel. The scene is composed of a (potentially large) number of three-dimensional geometric figures called *primitives*. Primitives can be as simple as a sphere or cube, or as complex as a fractal mountainside. Any light reaching the observer through

a given pixel must have come from the direction along a ray from the observer to the pixel in question. If one traces backward along this line of propagation into the scene, the surface from which the light was scattered can be discovered.

Since the physical properties of the surface are known, we can model the way it scatters light. The mathematical model used is called a shading model. Although the ray may intersect several objects in the scene, only the intersection point closest to the observer is relevant. Rays cast from the observer through the pixels are called primary rays. The brightness of each pixel on the viewplane is completely independent of its neighbors. Clearly, the intensity between adjacent pixels is highly correlated, but the calculations themselves are independent. Pixel independence gives ray tracing the fine-grained parallelism that makes it suitable for implementation on fine-, medium-, and coarse-grained parallel computers.

The intensity correlation between adjacent pixels makes a distributed ODB feasible in an indirect way. If primary rays are traced through the viewplane in a spatially coherent manner, then the ODB references generated will have a high degree of temporal and spatial locality within the ODB. This observation follows naturally from the functioning of the ray-tracing algorithm and the Kay ray-ODB intersection process [9]. This high degree of locality is just the property that makes an ODB cache feasible.

Though medium-grained distributed-memory parallel computers do not have sufficient memory per computing node to store very large ODB's, all is not lost. The database can be broken up, and the pieces stored on different nodes. Then, during the ray tracing process, parts of the database may be shuttled between nodes as needed. Each computing node does not necessarily need access to the whole ODB for every ray traced, but there is currently no easy way to predict which parts it will need access to. If we could easily predict which parts of the ODB certain rays needed access to, then the ODB distribution process could be greatly simplified. As it is, we can only guess at which pieces are needed based on previous accesses. This sort of guessing is exactly the function that a cache performs.

ODB Decomposition

In the Hypercube Ray Tracer, the ODB is organized into a hierarchy. Primitives are present only at the leaf

level. The body of the hierarchy is composed of *Hnodes* whose sole purpose is to impose an efficient geometrical structure onto the list of primitives which facilitates an efficient ray-primitive intersection process [9]. The structure is that described in [9], and the Hnode structure is generated according to [5].

We have chosen a dynamic ODB decomposition where primitives, rather than rays, are transmitted between nodes. Initially, the ODB is split evenly across the nodes, just as with Goldsmith's method [6]. The similarity ends here. The computing node to which a primitive is initially assigned is called its *home node*, and that node will always store a copy of the primitive. Once a computing node discovers that it does not have a part of the ODB it needs, that primitive is requested from its home node. This is called an *ODB miss*.

Primitives are retained on the nodes until their memory is exhausted and space is needed for another primitive. In this way, a node stores its share of the ODB plus some number of transitory primitives. Transitory primitives are discarded as needed to accommodate new transitory primitives needed in the intersection process. The set of transitory primitives is The least-recently-used (LRU) cache the cache. replacement method is used to select which transitory primitives are no longer needed. Since only the least recently used transitory primitives are thrown away, the more heavily used ones remain on the node. This greatly reduces the ODB miss rate, and hence message traffic between nodes. The ideal condition of having the whole ODB resident on each node is thus more closely approached.

This method of ODB decomposition has the ability to distribute a very large number of primitives across a number of computing nodes. Moreover, the ODB distribution is automatically adjusted to place the proper primitives just where they are needed. Much better performance is realized with this strategy than with a static ODB decomposition, and its fully-associative nature gives it an advantage over a direct mapped caching scheme [7].

Swapping Policy

Sending messages from one hypercube node to another is a costly process. There is a heavy time penalty to set up a message route plus a modest penalty for each byte transferred. In order to defray the high startup cost, long messages are preferred over short ones. A single primitive, the result of an ODB miss, would make a very short message. It is desirable to send several primitives at once when swapping is required. But which primitives should be picked? It would be most helpful to send additional primitives which are likely to be needed in the future. Indeed, the Kay ray-primitive intersection algorithm tests all of the children of a given Hnode at once. Therefore, it makes sense to send all siblings of the requested primitive as they will all be tested. Thus, we move from the

concept of swapping individual primitives to swapping all primitives associated with a given Hnode. This is analogous to the notion of line size in a conventional cache.

A number of special considerations in the hierarchy are required to support this ODB decomposition and swapping scheme. The hierarchy is composed of two basic entities: the group of Hnodes which comprise the interior of the hierarchy, and the primitives. The Hnodes are only responsible for about 14.3% of the total number of nodes in the ODB [2]. Further, Hnodes takes only one third the memory space to store as Therefore, the Hnode infrastructure primitives. effectively is responsible for only about 5% of the size of the ODB. Thus, each node can easily store the ODB infrastructure and just swap groups of primitives. This also allows the Kay algorithm to go all the way to the leaf level before an ODB miss is possible. This requires each Hnode to contain information about whether or not its child primitives are resident.

Hnodes must also keep track of the LRU reference word for cache replacement purposes. The Hnode structure contains the following information.

TABLE 1
Fields in the Hnode Data Structure

Field and Description

- 1. Pointers to sub-hierarchies (max 8).
- A unique ID number.
- 3. A bounding volume enclosing all sub-hierarchies.
- LRU reference word.
- A flag which is true if this Hnode's child primitives are not resident.

Note that not all Hnodes have child primitives. Some Hnodes will have only other Hnodes as children. These interior Hnodes are *unswappable*, and do not take part in the ODB distribution process. The balance of the Hnodes are called *swappable Hnodes*, and do take part in the distribution process. Stated another way, an Hnode is swappable if and only if at least one of its children is a primitive.

As stated above, a certain portion of the ODB must remain resident on each computing node. Rather than thinking of this portion as a set of primitives, we shall think of it as a set of swappable Hnodes. The swappable Hnodes are divided evenly among the processors rather than the primitives directly. In this way, the child primitives of a swappable Hnode are never split between two computing nodes. In a scene with a large number of primitives, the unevenness in the distribution of primitives caused by this method is negligible. Hnodes are assigned to computing nodes in a round-robin fashion as their ID numbers order them. This randomizes the ODB's initial distribution, and helps to even out the burden of ODB requests.

Primitives are swapped in and out as groups. The LRU replacement algorithm targets the Hnode whose LRU reference word is smallest for replacement. All

child primitives of the target Hnode are freed, and the Hnode is marked as *swapped*. The targeting and freeing operations are repeated until enough space is available for the incoming primitives.

The Ray Tracing Loop

Now that ODB distribution has been addressed, we must now address the problems this causes in the ray tracing loop. Since parts of the ODB can be missing on each node, the Kay intersection algorithm may fail. When it does fail, a request for the missing primitive must be formulated and sent to the primitive's home node. (A primitive's home node is based on the unique ID number assigned to the Hnode parent of the primitive.) Once the requested primitive is received and inserted into the ODB, we must restart the intersection algorithm from where it stopped. This prevents thrashing, but requires paying a considerable price in terms of program complexity.

Once an ODB miss occurs, what happens while the node is waiting for primitives from another node? That node may do one of two things: wait for the primitive to be sent from another node, or work on another ray. Considering the cost of sending a message to another node, and waiting for it to reply, waiting is out of the question. Therefore, the node must occupy its time doing something constructive; processing another ray is an ideal choice. Therefore, the entire state of the ray tracing process must be saved when an ODB miss occurs. The ideal place to save this information is in the same data structure as the offending ray so that the ray's entire context is neatly localized.

Now that intersection may be stopped and restarted, we must consider another step in the ray tracing process, namely the shading step. The shading model casts shadow rays every time it is evaluated, and optionally casts reflected and refracted rays. These secondary rays must also be ray-traced. Since they may also cause ODB misses, the shading model evaluation must be made interruptible, too! To complicate matters further, the shading model may be interrupted in no less than three different locations: once for each light source when casting a shadow ray, once for the reflected ray, and once for the refracted ray! Now, the ray tracing loop has become a very complex choreography of interruptible states, spawning of sub-rays, and resumption of control. The following finite-state automaton (FSA) is our solution to the control problem (Figure 2). In Table 2, we see that rays are divided into a number of different types: primary rays, secondary rays, and shadow rays. The only difference between the types is the way in which the shading model operates. For primary rays, the full shading model is evaluated, and the result is stored at the appropriate pixel coordinates in the local frame buffer. Secondary rays execute the full shading model, but pass their intensity to their parent ray rather than the frame buffer. Shadow rays need not be shaded at all, only intersected with the

ODB.

TABLE 2 Ray States

State and Description

- Ready to intersect A ray is set up and ready be be intersected against the ODB.
- Pending object from another node A ray has suffered an ODB miss, and is waiting for the required primitives to be sent from elsewhere.
- Shadow ray setup Shadow rays are set up and spawned from this state. A ray to a different light source is spawned each time this state is entered.
- Pending on shadow ray Once a shadow ray has been spawned, the parent ray must wait for it to complete.
- Process shadow ray The result of the shadow ray intersection are stored and control is passed back to the "shadow ray setup" state to cast more shadow rays.
- Shading This step in the shading model performs all operations that depend only on the results of the shadow rays. i.e. ambient, diffuse, and specular components.
- Reflective shading If the surface of the primitive in question is reflective, this state spawns a reflected ray.
- Pending reflected ray Control comes here to wait on a reflected ray to be traced.
- Process reflected ray The contribution of the reflected ray is added into the overall shading in this state.
- Transmissive shading If the surface of the primitive in question is transmissive, this state spawns a refracted ray.
- Pending transmitted ray Control comes here to wait on a refracted ray to be traced.
- Process transmitted ray The contribution of the refracted ray is added into the overall shading in this state.
- 13. Forward results Ray results are ready to be passed on. Depending on the ray type, the results are either put in the local frame buffer (primary ray), or forwarded to the parent ray (shadow or secondary ray).

TABLE 3 Ray State Transition Events

Type and Description

- ODB miss This event is posted by the "ready to intersect" state when an ODB miss occurs.
- Object received This event is posted when primitives arrive from another node as the result of an ODB miss.
- Spawn Posted whenever a state had to spawn a subray. This happens for shadow rays, reflected rays, and refracted rays.
- Complete This event is posted to a parent ray when a child ray has completed.
- Done Posted by state action functions, this event signals that the state completed successfully, and the ray is ready to move on to the next state.

- 6. Backtrack If a shadow ray intersects a transparent object, it is not necessarily occluded. Used to restart the intersection process to find the next intersection point along the shadow ray.
- 7. Missed If the intersection process misses all objects in the ODB, this event shortcuts straight to the "Forward Results" state.

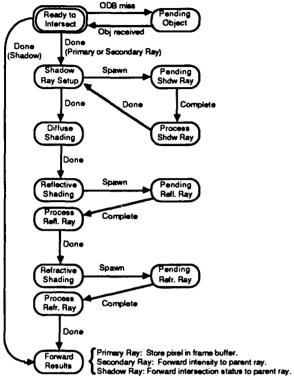


Figure 2: Control Flow for Rays

Transitions between ray states are caused by events posted to a specific ray during the ray-tracing loop (Table 3). These events are based on the result of the action associated with each state. Actions perform the various steps in the ray tracing process. For example, the action associated with the first state in the FSA is to try to intersect the ray with the ODB. If the intersection fails, the action function posts an ODB miss event for the ray, and terminates. The result of this event is to place the ray in the Pending Object from Another Node state. If the intersection succeeds, the action function posts a Done event and terminates. The result of this event is to place the ray into the Shadow Ray Setup state. The concept of state driven ray tracing complicates the classical ray tracing loop, but divides it into an interruptible series of modular operations. Although some of the states presented above could be merged, they are left separate for clarity.

As stated earlier, multiple rays are used so time is not wasted waiting for ODB misses to be resolved. Since there can be a number of pending rays equal to a preset maximum recursion depth, a way is needed to keep track of all of these rays. A way is also required to keep track of events destined for a particular ray. The solution used here is a ray queue to keep the rays, and an event queue to keep track of the events.

As new rays are created, they are pushed onto the ray queue to begin their journey through the states that will ray trace them. Similarly, ray-event tuples are pushed onto the event queue for evaluation. A scheduler is responsible for driving the FSA from the rays and events. The scheduler sits at the top of the control structure for the new node ray-tracing loop. Algorithm 1 gives the pseudocode representation for the node program. One will notice the striking resemblance between Algorithm 1 and any standard round-robin task

```
Download ODB from the host
While there are pixels to ray trace and ray
queue not empty
   /* Service queued events. */
   While event queue not empty
       Pop event queue
       Determine next state of ray
   Endwhile
   /* Service prim. requests */
   /* from other nodes.
   If ODB request from other node
       Pack requested portion
       Send it to requesting node
   Endif
   /* ODB Request Reply */
   If there is ODB request reply
       Receive the message
       Unpack it into local ODB
       Notify all rays pending
   Endif
   /* Add a new primary ray */
   If there is room on ray queue
       Construct a new primary ray
       Push it onto the ray queue
   Endif
   /* Execute a ray's state
   /* function.
   Pop a ray from ray queue
   Execute its state function
Endwhile
Send local frame buffer to host
```

Algorithm 1: Scheduler for State Driven Node Program

scheduler. In the ray tracer's case, the analog for a process is the ray.

Image Decomposition

When the leap is made from a duplicated ODB to a distributed ODB (DODB), many things change. The structure of the ODB changes from a fully intact hierarchy to a hierarchy missing most of its leaves. The hierarchy nodes themselves become more complex. Ray-ODB intersection becomes an interruptible, reentrant process rather than classical straight-line code. Even the ray tracing loop itself changes from a

regimented and easy-to-understand loop into a complex scheduler driving a thirteen state FSA.

After such a drastic change to the basic ray tracing loop, the suitability of the standard image decomposition needs to be reassessed. Using the "comb" decomposition, the image plane is divided into equal-area pieces, and each piece assigned to a computing node for ray-tracing [2]. Within the assigned area, which is usually rectangular, pixels are ray-traced from the upper left corner toward the bottom right corner by rows. There is one basic problem associated with this decomposition - that of locality. choosing a different image decomposition, we may reduce the number of ODB misses considerably. If the DODB is to perform well, then the rays tested against it should be fairly localized with respect to their positions and directions within the scene. This locality of reference keeps the number of ODB misses down, and the performance up. If widely varying rays are intersected against the DODB, then there will be a much higher miss rate, and correspondingly lower performance. Experiments verify the lower performance of the standard decomposition, and show a very poor load balance. It is therefore desirable to invent a new image decomposition to solve the load balance and locality problems simultaneously.

The solution used by the Hypercube Ray Tracer is what is generally called the "block" decomposition. The image plane is divided into a large number of small rectangular blocks which are assigned dynamically to processors. Each block encloses pixels that one node will be responsible for ray tracing. These blocks are small enough such that all rays passing through it can be considered coherent. When a computing node finishes ray tracing all of the pixels in its block, then a new block is assigned which is spatially close to the previous one. In this method, the dynamic block assignment solves the load balancing problem, and new blocks are chosen close to old blocks to give heightened locality of reference in the ODB.

Block assignments are managed by a separate program running on the iPSC/2's System Resource Manager (SRM). As computing nodes complete their blocks, they send the block's frame buffer to the SRM where is is copied into the final image. After this is done, the SRM assigns a new block to the node as close to the old one as possible. This process continues until all blocks have been ray-traced.

Results

Figure 3 show the number of ODB requests received by all 32 nodes during the course of ray-tracing a single image of 4208 primitives. In this case, the cache size amounted to some 10% of the size of the whole ODB. This figure shows the evenness of the ODB distribution across the nodes. No one node, or set of nodes, are bearing the brunt of ODB swapping traffic.

ODB Requests per Node for Bar Image (Cache alze = 10%)

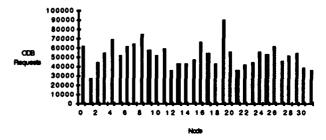


Figure 3: ODB Requests

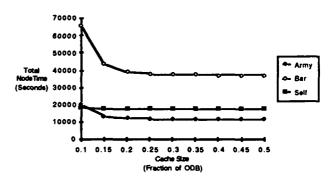


Figure 4: Ray Tracing Time

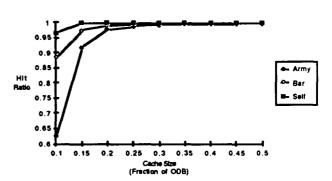


Figure 5: Cache Hit Ratio

In Figure 4, we see the effects of reducing the cache size for three different images. The "Bar Image" contains 4208 primitives, the "Self Portrait" image contains 1410 primitives, and the "Snowman Army from Hell" image contains 1310 primitives. Specifically, Figures 4 shows total ray-tracing time versus cache size and Figure 5 shows cache hit rate versus cache size. Note that total ray-tracing time is the sum of the ray-tracing times for all 32 nodes of the iPSC/2. In all cases, the cache hit rate remains very high, and the ray-tracing time remains relatively constant until a cache size of roughly 20% is reached. Below a cache size of 20%, the ray-tracing time increases rapidly with the decreasing cache hit rate. One will note that the knee of the total ray-tracing time curve is at a slightly lower cache size than the knee of the hit rate curve. This effect is caused by the now

interruptible nature of the ray-tracing loop. When an ODB miss occurs, the offending ray is placed on the ray queue, and another ray is initiated. Even though the cache performance decreases, the ray-tracing speed does not! Not shown is the dependence of performance on ray queue size. A slightly larger ray queue makes a marked difference in performance for small cache sizes.

As we can see from the performance figures, the Hypercube Ray Tracer's ODB distribution strategy is very effective for cache sizes down to only 15% of the total ODB size. Furthermore, the cache implements a fully-associative, self-balancing structure that requires no preprocessing to set up. The fully-associative nature of the cache insures that no heavily-used primitive groups are replaced just because they happen to lie in the same set as another transitory primitive. We must not forget that the only reason that such a caching scheme can enjoy any success at all lay in the temporal coherence with which the ODB is queried by the rayobject intersection process. This temporal coherence is due directly to the fact that rays are traced in close proximity to one another on the viewplane. If the rays were traced randomly throughout the viewplane, then no amount of caching would improve performance.

Conclusions and Future Work

We can also see that this caching scheme scales up well with the number of nodes, and particularly well with the ODB size. Also, since there are no artificial boundaries imposed on the ODB cache, it takes maximum advantage of node memory, duplicating those portions of the ODB which are used heavily.

As with all things in life, there is a price for such progress, and the Piper's name is Complexity. Since an ODB miss can happen at any point during the ray-tracing process, the state of each ray must be saved until such time as it can resume. But since the ray-tracing process itself is so time-consuming, this additional complexity imposes little to no performance penalty.

A number of elements in this distributed ODB scheme warrant further investigation. One is the initial construction of the hierarchy. The ODB hierarchy is the one fixed data structure remaining in the Hypercube Ray Tracer. Presently, it is constructed to minimize the total surface area of the bounding volumes of all subhierarchies. This policy makes a considerable performance difference with respect to a blindly constructed hierarchy. However, it leads to primitives being placed near the root of the hierarchy. It is unclear just how much this ragged hierarchy structure impacts the performance of the distributed ODB.

Rays could be swapped across nodes as well as primitives. This would be more economical in cases where a large number of ODB misses would occur. Instead of shipping all of the nonresident primitives to a node, the node would have the option of shipping the offending ray to the primitives.

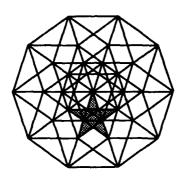
Perhaps most tantalizing would be a dynamic

restructuring of the entire ODB hierarchy to take more advantage of ray coherence. Presently, the Kay intersection algorithm takes no advantage of ray coherence since the structure of the ODB is fixed. If a scheme could be devised to restructure the ODB toward the goal of decreasing the number of tests performed by the Kay algorithm, then the performance of the distributed ODB cache would be improved as well as absolute ray-primitive intersection time.

A short-term goal is to investigate the effect of a least-frequently-used (LFU) cache replacement policy. A longer-term goal might be to include secondary storage in the ODB distribution scheme. A strategy of this sort would make possible the ray-tracing of scenes of well into the millions of primitives.

Bibliography

- [1] Arvo, James, and David Kirk (1987) Fast Ray Tracing by Ray Classification, *Computer Graphics*, 21(4), July 1987, pp. 55-64.
- [2] Carter, Michael B. (1989) Ray Tracing Complex Scenes on a MIMD Concurrent Computer, Masters Thesis, Oklahoma State University.
- [3] Fujimoto, Akira, T. Tanaka, and K. Iwata (1986) ARTS: Accelerated Ray Tracing System, *IEEE Computer Graphics and Applications*, 6(4), April 1986, pp. 16-26.
- [4] Glassner, Andrew S. (1984) Space Subdivision for Fast Ray Tracing, *IEEE Computer Graphics and Applications*, 4(10), Oct. 1984, pp. 15-22.
- [5] Goldsmith, Jeff, and John Salmon (1987) Automatic Creation of Object Hierarchies for Ray Tracing, IEEE Computer Graphics and Applications, 7(5), May 1987, pp. 14-20.
- [6] Goldsmith, Jeff, and John Salmon (1988) A Hypercube Ray-tracer, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3), Vol. 2, Jan. 1988, pp. 1194-1206.
- [7] Green, S. A., and D. J. Paddon (1989) Exploiting Coherence for Multiprocessor Ray Tracing, *IEEE Computer Graphics and Applications*, 9(6), November 1989, pp. 12-26.
- [8] Kajiya, James T. (1983) New Techniques for Procedurally Defined Objects, Computer Graphics, Vol. 17, No. 3, July 1983, pp. 91-102.
- [9] Kay, Timothy L., and James T. Kajiya (1986) Ray Tracing Complex Scenes, ACM SIGGRAPH 1986, 20(4), August 1986, pp. 269-278.



The Fifth Distributed Memory Computing Conference

9: Sorting

Parallel Sorting on Symult 2010

P. Peggy Li
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Yu-Wen Tung*
USC - Information Science Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

Abstract

In this paper, three sorting algorithms, Bitonic sort, Shell sort and parallel Quicksort are studied. We analyze the performance of these algorithms and compare them with the empirical results obtained from the implementations on the Symult Series 2010, a distributed-memory, message-passing MIMD machine. Each sorting algorithm is a combination of a parallel sort component and a sequential sort component. These algorithms are designed for sorting M elements of random integers on a N-processor machine, where M > N. We found that Bitonic sort is the best parallel sorting algorithm for small problem size, (M/N) < 64, and the parallel Quicksort is the best for large problem size. The new Parallel Quicksort algorithm with a simple key selection method achieves a decent speed-up comparing with other versions of parallel Quicksort on similar parallel machines. Although Shell sort has a worse theoretical time complexity, it does achieve linear speedup for large problem size by using a synchronization step to detect early termination of the sorting steps.

Introduction

As indicated by Knuth in his famous book on sorting and searching [1]:

It would be nice if only one or two of the sorting methods would dominate all of the others, regardless of the application or the computer being used. But in fact, each method has its own peculiar virtues.

This remains true, if not more so, for sorting algorithms on parallel machines for two reasons. First, the performance of a parallel sorting algorithm depends on the degree of parallelism it can exploit on a

given architecture. For example, it often makes difference on level of parallelism one could exploit on an SIMD and on an MIMD machine. Second, the performance also depends on the speed of certain critical operations the underlying parallel machine could deliver. For example, interprocessor communication could be a dominating operation for distributedmemory machines because parallel sorting algorithms often require the same order of magnitude of communication steps as that of computation.

In this paper, we focus on only a class of MIMD machine on which the issue of interconnection network is not very important, and the communication speed is nearly balanced with the computation speed. By choosing such seemingly general-purposed, yet real, machine, we are able to concentrate on finding which sorting methods, or combinations of sorting methods, are possibly among the fastest on an MIMD machine.

We shall also confine ourselves to sorting a long list of random input data using less number of processing elements (or nodes). That is, the sorting problem we are interested in is to sort M elements of random integers on an N-node MIMD machine, where M>N. Initially, M unsorted elements are evenly distributed to each computation node. Each node operates on its own set of data independently, but can send or receive data from another node. When all nodes terminate, each node should hold a chunk of sorted list, and chunks are stored in consecutive order across all nodes such that the smallest chunk is stored in the first node and so on. Chunk size may or may not be M/N depending on the algorithm used.

Because of the problem nature M > N, each of the three sorting algorithms we have implemented is a combination of parallel sort (across nodes) and sequential sort (for local list). We used a parallel version of Quicksort [2], Batcher's bitonic sort [4], and a mixture of Shell's sort and odd-even transposition sort [1] as our parallel sorting strategies, and the UNIX/BSD qsort routine as the sequential sorting method. For simplicity, we shall call our algo-

^{*}Supported by NASA Cooperative Agreement NCC-2-539 and RADC contract F30602-88-C-0135

rithms Bitonic sort, Shell sort and parallel Quicksort, respectively, in the following text.

Quicksort is not only a fast sequential sort method, it is also a parallel method by its divide-and-conquer nature. The only potential problem with the efficiency of a parallel Quicksort is the selection of its splitting keys. If such keys are randomly selected, the input list can be divided into uneven sublists and cause load unbalancing. Carefully calculated splitting keys will solve this problem but the extra calculation becomes a cost itself. So an efficient implementation needs to strike a balance between two extremes, which is, fortunately, not very hard to achieve. Impressive results for parallel Quicksort have been reported for a vector machine CDC STAR [3] and hypercube-interconnected MIMD machines [5], among others.

Batcher's bitonic sort [4], on the other hand, has been widely used across almost all kinds of parallel computers – sorting networks, hypercube machines [6], two-dimension mesh machines [9], SIMD machines [7] for its simplicity and stability. It has a time complexity $O(\log^2 M)$ for sorting M elements, which is reasonably efficient. The less known Shell sort is also selected because it appears to be a very efficient algorithm when implemented on Caltech/JPL's Hypercube machine [5].

In the rest of the paper, we will first introduce the underlying machine we used in our study, and its computation and performance model; followed by the three sorting algorithms and their time complexities. Then, we will discuss our empirical performance result, and address a few related issues such as how general our result can be, and what other sorting methods may also be considered.

Computation and Performance Model

The Symult Series 2010 system (S2010) is a distributed-memory message-passing MIMD computer consists of up to 1024 computational nodes interconnected by a high speed message-routing network (GigaLink). Each computational node has a Motorola MC68020 microprocessor as its CPU, operating at 25 MHz and augmented by the Motorola 68881 floating-point co-processor. A SUN-3 workstation is used as the front-end computer. The operating system on the S2010 nodes is called Reactive Kernel, and the programming environment on the front-end computer, serving as the interface between the users and the S2010, is called Cosmic Environment [8].

S2010 is facilitied with a fast communication network, called GigaLink network. A custom-designed

message routing chip — Automatic Message Routing Device (AMRD) — provides fast fixed-route point-to-point message routing using "worm-hole" routing algorithm. The interprocessor communication rate is 13MB/sec regardless of the distance between source and destination. This feature makes the S2010 resemble to a fully-connected machine.

To characterize the machine behavior, we carefully measured timing for many computation and communication instructions. Here are some of the timing results useful for our sorting analysis, where one integer is equivalent to four bytes:

- copy one integer from one memory location to another, without taking memory allocation overhead into account, takes about 0.45 μs;
- memory allocation overhead per memory copy function (bcopy) is about 8 μs;
- comparison-exchange for two integers takes about 6.8 μs;
- transmitting one integer in a typed message from one node to another, without taking overhead into account, takes about 0.31 μs under low to normal traffic load.
- average overhead for sending a typed message from one node to another takes about 251 µs.

In other words, if routing a message with size K (integers) takes time $T_{route}(K)$, copying a same size message locally takes time $T_{copy}(K)$, and performing compare-exchange on K integers takes time $T_{comp-ex}(K)$, then

$$T_{route}(K) = T_{route-overhead} + T_{route-int} \cdot K$$

$$= 251\mu s + 0.31\mu s \cdot K$$

$$T_{copy}(K) = T_{copy-overhead} + T_{copy-int} \cdot K$$

$$= 8\mu s + 0.45\mu s \cdot K$$

$$T_{comp-ex}(K) = 6.8\mu s \cdot K$$
(1)

So we could conclude that, on S2010, the overhead for each message send/receive is very large, but the transmission speed is comparable to that of memory access speed. Therefore, in our sorting algorithms, the interprocess communication is a dominating term when M/N is small, and it graduately reduces its effect when M/N gets larger. Assuming the number of compare-exchange steps is the same as the number of messaging, then the communication overhead (i.e. $T_{route}(K)/T_{comp-ex}(K)$) is 41.4% for K=100 and drops to 8.2% for K=1000.

In the following timing analysis, we shall assume that each element to be sorted is represented as an integer, for simplicity. Thus K means the number of elements in each step of computation.

The sequential *qsort* routine takes an important role in all three algorithms, it has a time complexity $O(K \log K)$ for a single S2010 node to sort K elements. By experiments, we found that the timing equation for *qsort* with random input data can be represented as follows:

$$T_{qsort}(K) = O(K \log K) = 8.5 \mu s \cdot K \log K$$
 (2)

Bitonic Sort

In our implementation of this algorithm, the machine is configured as a N-node hypercube. Initially each node has M/N unsorted elements. Each node first sorts its data internally using the qsort routine, and then performs $(\log N \cdot (\log N + 1))/2$ steps of compare-exchange operation along all dimensions of the cube. After running the algorithm, every nodes have M/N elements sorted both locally and globally.

Our algorithm for each individual node is shown below, where dim, my_nid, and mask are the dimension of the cube, the node id, and a mask flag for selection of nodes, respectively:

 Sort the (M/N) elements locally in each node using a qsort. Sort in ascending order if my_nid is even, in descending order if my_nid is odd.

For i := 0 to (dim - 1) step 1 do (2), (3), and (4)

- If the (i+1)-bit of my binary address is 1, mask := 1; otherwise, mask := 0.
- 3. For j := i to 0 step -1 do

 (a). exchange my (M/N) elements with my jth bit neighbor; (b). compare/exchange the two
 lists and copy the smaller half into the data area
 if mask = the j-th bit of my binary address; copy
 the larger half into the data area otherwise.
- 4. Locate the maximum (or minimum) of the bitonic sequence in each node and perform a merge on sublists of length M/N. The sorted sublist is in ascending order if mask = 0; otherwise, it is in descending order.

Since each node has M/N elements, the time complexity of step (1) is $O(\frac{M}{N}(\log \frac{M}{N}))$. Each compare-exchange iteration in (3) takes time $2T_{route}(\frac{M}{N}) + T_{comp-ex}(\frac{M}{N}) + T_{copy}(\frac{M}{N})$, and there are $(\log N(\log N + 1))/2$ iterations in total. Each merge operation in step (4) takes time $T_{comp-ex}(\frac{M}{N} + 2\log \frac{M}{N})$, and this operation is performed $\log N$ times.

As a result, the total time for Bitonic sort, based on our timing equations (1) and (2), can be expressed with unit time μs as follows:

$$T_{Bitonic} = O(\frac{M}{N}(\log \frac{M}{N})) + (\log N(\log N + 1)/2) \cdot (510 + 7.87(\frac{M}{N})) + 6.8 \log N(\frac{M}{N} + 2 \log \frac{M}{N})$$

$$= 8.5 \frac{M}{N}(\log \frac{M}{N}) + (241.4 + 3.94 \frac{M}{N}) \log^2 N + (13.6 \log M + 10.74 \frac{M}{N} + 255) \log N$$
(3)

The empirical timing curve for N = 16 is shown in figure 1.

Shell Sort

As described earlier, here Shell sort means a method that combines Shell's method and odd-even transposition sort as internode sort, and qsort as sequential sort. This algorithm, as well as the parallel Quick-sort algorithm, are to be executed on a ring topology – the sorted data will be stored in the same way as in the Bitonic sort case, but with a slight difference that node address is arranged in ring topology. Both hypercube and ring topologies can be easily configured on the S2010, without significant performance difference.

This algorithm has three steps:

- 1. Sort (M/N) elements locally in each node with qsort.
- 2. Do a compare-exchange operation between pairs of adjacent nodes along the i-th cube dimension, for $i = \log N 1, \ldots, 0$.
- Do compare-exchange operations between pairs of adjacent nodes in the ring topology until no exchange is made in all the node.

The first part is a Shell's sort except that only one compare-exchange operation is performed in each hypercube dimension and the result list is partially sorted. This part takes $\log N$ compare-exchange operations in total. The second part is an odd-even transposition sort which terminates when no data is exchanged in all the node.

The number of odd-even transposition steps is equal to the maximal distance of a mispositioned element to its sorted position. After the diminishing-increment steps, the worst-case maximal distance is $(N-2\sqrt{N}+1)$, where N is the number of nodes.

Given an arbitrary element a, assuming that y and xare the addresses of the nodes that a is located after step (1) and after sorting, respectively. Thus, |y-x| is the number of odd-even transposition steps required to move a to its final position. Let a be such an element that has maximal |y-x| and x < y, now we like to find the minimal x for a given y. Let the binary address of y be (y_1, y_2, \ldots, y_d) , where $d = \log N$, i.e., the dimension of the cube. After step (1), all the elements in the nodes whose addresses can be derived from y by changing one or more y_i 's from 1 to 0 should be smaller than the elements in y. If there are k "1" bits in the binary address of y, there will be at least $(2^{k}-1)$ nodes in which the elements are smaller than those in y after step (1). Thus, the minimal element that may be located in node y after step (1) is always greater than the elements in the first $(2^k - 1)$ nodes after sorting. In other words, the minimal a in node y will be stored in the 2^k -th node $(x = 2^k - 1)$ after sorting is done. To maximize |y-x|, we shall find the maximal y with a proper k. Obviously, the maximal y which has k "1" bits is the one having all 1's in the most significant bits and $y = N - 2^{(d-k)}$. So $(y-x) = (N-2^{(d-k)}-2^k+1)$ and the maximal (y-x) is equal to $(N-2^{d/2+1}+1)$ or $(N-2\sqrt{N}+1)$ when k = d/2.

Therefore, in the worst case, there are $(N-2\sqrt{N}+1)$ compare-exchange operations in step (3). Each compare-exchange operation in step (2) and (3) takes the same time as one iteration in Step (3) of Bitonic Sort, i.e. $2T_{route}(M/N) + T_{comp-ex}(M/N) + T_{copy}(M/N)$. Therefore, the worst case time complexity of the Shell sort is

$$T_{Shell} = 8.5 \frac{M}{N} \log \frac{M}{N} + (N - 2\sqrt{N} + \log N)(510 + 7.87 \frac{M}{N})$$
 (4)

The first term is the time for sequential sort of the local M/N element sublist. The real timing for the case N=16 is shown in figure 1.

Parallel Quicksort

The quicksort is a divide-and-conquer sorting algorithm which is potentially applicable to parallel computation. In order to get the best performance of the quicksort, the splitting keys should be selected with great care so that the list to be sorted can be decomposed into two sublists of equal length. This fact is even more important in the parallel quicksort because the improper selection of the splitting keys results in load imbalance and the computation time

is determined by the slowest node.

Similar to Bitonic sort and Shell sort, the unsorted list is stored evenly across the cube initially, i.e., each node has M/N elements in arbitrary order. After sorting, the sorted list will be stored in the cube in consecutive order but each node may have different number of elements. The parallel quicksort works as follows: First, (N-1) splitting keys are selected using a presorting algorithm. Second, the list in each node is split into two parts according to a proper splitting key and exchanged with its neighbor along a certain dimension. This splitting process repeats log N times. At last, each node sorts its local list with a fast sequential sorting algorithm.

The algorithm and its time-performance are described as follows:

- 1. Choose k samples randomly from the $\frac{M}{N}$ -element sublist of each node. Find the largest and the smallest elements in the sample, let them be (max, min).
- 2. Perform the maximum and minimum operations on each node's (max, min) pair globally across the nodes to find the maximum and the minimum elements, say (gmax, gmin), in the whole sample. This global operation is done in a binary tree manner, and it needs $\log N + 1$ communication steps, i.e., $(\log N + 1) \cdot (T_{route}(2) + T_{comp-ex}(2))$.
- 3. Equally divide (gmax gmin) into N 1 intervals and use the N boundary elements as the splitting keys. Since each node only needs $\log N$ splitting keys, each node can use a binary search to find all its keys in $\log N$ steps.
- 4. for i := dim 1 to 0 step -1 do compare my sublist with the i-th splitting key and divide it into two sublists. if (my_nid < neighbor[i]) exchange the larger sublist with the smaller sublist of my i-th bit neighbor else exchange the smaller sublist with the larger sublist of my i-th bit neighbor</p>
- 5. Sequential sort the sublist locally.

endif.

The main part of the parallel Quick sort, i.e, step (4), takes

$$\log N \cdot (2T_{route}(\frac{M}{N}) + T_{comp-ex}(\frac{M}{N}))$$

for the best case, assuming each node always hold M/N elements after each compare-splitting step.

With a good set of splitting keys, the parallel Quicksort has a best/average time performance:

$$T_{Quich} = 8.5 \frac{M}{N} \log \frac{M}{N} + 7.41 \frac{M}{N} \log N + 767.2 \log N$$
 (5)

The sampling time in step 1 is proportional to the sample size in each node, which is negligible. The first term of the equation is the sequential sorting time in step (5). And the real timing in the case of N=16 is shown in figure 1.

Performance Comparison and Analysis

We have measured execution time for each of the above three sorting algorithms for N=8, 16, 32 and 64, and M ranges from 2^6 to 2^{19} . Figure 1 shows the timing curves with the execution time versus $\log M$ for N=16.

Figures 2 and 3 are speedup curves calculated from real execution time of the three parallel algorithms for N = 16 and 64, respectively.

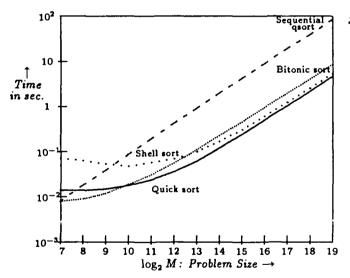


Figure 1. Timing on a 16-node S2010

Input lists are generated by using UNIX random routine. The execution time is determined by the sorting time of the slowest node. The down-loading and up-loading (input and output) time is not considered in our experiments.

From these speed-up curves, it is observed that the increasing communication overhead degrades the sorting speed of small lists (for lists with 1K elements or less) when the machine size increases. In the case of Bitonic sort, which has the lowest communication overhead and is the fastest algorithm for small lists,

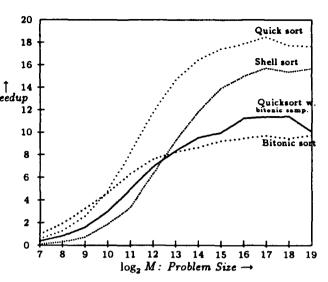


Figure 2. Speedup curves on a 16-node S2010

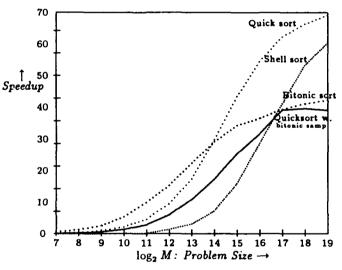


Figure 3. Speedup curves on a 64-node S2010

the ratio of the communication time to the computation time is more than two for small lists.

In the case of Shell sort, the timing equation 4 does not include the time to broadcast the boolean flag which indicates if any exchange has been made in each compare-exchange step in (3) of the Shell sort algorithm. This value may be negligible when $M\gg N$, but becomes the major overhead when N gets large, or when $M\simeq N$. Broadcasting is done in binary tree manner which requires $(\log N+1)$ steps of message transmission after each compare-exchange step. For the worst case, the broadcast overhead is as high as $253.3(N-2\sqrt{N}+1)(\log N+1)$. Although the parallel sorting part of T_{Shell} has an O(M) time complexity in the worst case, the broadcasting step may save a lot of

compare-exchange steps for random data input. The empirical result shows that the parallel Shell sort can achieve linear speed-up for large problem size random data. See figures 2 and 3.

As to parallel Quicksort, empirical result shows that a presorting procedure as simple as the aboved-mentioned splitting key selection mechanism can result in very good load balancing, thus a super linear speedup is observed. A more complicated presorting algorithm based on the bitonic sort has also been attempted, but it results in a higher overhead, i.e., $O(k \log^2 N)$, for k samples each node, and a worse load balancing than the above algorithm. Consequently, we can conclude that for random data, the simple equally-divided key selection method can achieve the best performance of the parallel Quicksort. See figures 2 and 3.

Unlike the other two sorting algorithms, the sorted list obtained from this algorithm is not evenly distributed in each node. This is not a problem if the sorted list is up-loaded to the host machine without further computation. On the other hand, if sorting is just a part of the computation and the sorted list needs to stay in the cube for later use, the unbalanced data distribution may not be desirable. In this case, we may need to rearrange the elements so that each node keeps the same number of elements. The cost for the redistribution needs further investigation.

Conclusions

We have implemented three sorting algorithms on S2010, a distributed-memory message-passing MIMD machine. These algorithms are chosen because they can be parallelized easily on a mesh or hypercube architecture. Each sorting algorithm is a combination of parallel and sequential sorting methods and has a different time complexity.

In the parallel sorting component, Bitonic sort takes a fixed number of steps to sort despite of the input data pattern, with time complexity $O(\frac{M}{N}\log^2 N)$. Parallel Quicksort has a performance that depends on how good splitting keys are selected, and it is shown that with a little overhead of presorting this algorithm can achieve very good load balancing, and thus a best time performance $O(\frac{M}{N}\log N)$. The performance of the Shell sort is constrained by its second part, the odd-even transposition sort, which is a slow sequential sorting algorithm. Nevertheless, by taking the advantage of the asynchronous nature of S2010, the parallel version of the Shell/odd-even transposition sort can be as good as the parallel Quicksort when

 $M\gg N$.

In the sequential sorting part, which is performed on the M/N elements locally on each processor as the first step in Bitonic sort and Shell sort, or on varied number of elements locally as the last step in parallel Quicksort, has a time complexity $O(\frac{M}{N}\log\frac{M}{N})$.

The overall performance of the three algorithms is a combination of this sequential performance and the parallel sort performance. We found from our empirical results, for relatively small size of problems, M/N < 64 say, Bitonic sort is the best because it has the lowest synchronization overhead in the algorithm. The parallel Quicksort is the best for large problem size, which agrees with our analysis. It is interesting to learn that Shell sort outperforms Bitonic sort in the case of large problem size, which is mainly due to the fact that the Shell sort often terminates the sorting steps earlier. Both the parallel Quicksort and Shell sort achieve linear speed-up comparing to sequential quort for large problem size on 8 to 64 processor machines. Shell sort is the slowest among all for the small problem sizes because of its high synchronization overhead.

References

- [1] Knuth, D.E., The Art of Computer Programming, Vol.3, Sorting and Searching. Reading, MA: Addison-Wesley, 1973.
- [2] Hoare, C.A.R., "Quicksort," Computer J., vol. 5, no. 1, 1962, pp.10-15.
- [3] Stone, H.S., "Sorting on STAR," IEEE Trans. on Software Engineering, Vol.SE-4, No.2, March 1978.
- [4] Batcher, K.E., "Sorting Networks and Their Applications," in Proc. AFIPS 1968 SJCC, vol. 32, Montvale, NJ, AFIPS Press, pp.307-314.
- [5] Felten, E., Karlin, S. and Otto. S., "Sorting on a Hypercube," unpublished Caltech report Hm-244, 1986.
- [6] Johnsson, S.L., "Combining Parallel and Sequential Sorting on a Boolean N-Cube," Proceedings of the 1984 International Conference on Parallel Processing, 1984, pp.444-448.
- [7] Tung, Y.-W. and Mizell, D.W., "Two Versions of Bitonic Sorting Algorithms on the Connection Machine," Third Annual Parallel Processing Symposium, Fullerton, California, March 1989.
- [8] "Programmer's Guide to the Series 2010TM System Preliminary," symult Systems Corp., January 1989.
- [9] Nassimi, D., and Sahni, S. "Parallel Permutation and sorting algorithms and a new generalized connection network," JACM 29(3), 642-667, 1982.

Load Balanced Sort on Hypercube Multiprocessors

Bülent Abalı IBM T.J. Watson Research Center Füsun Özgüner and Abdulla Bataineh
The Ohio State University
Dept. of Electrical Engineering

Abstract

A parallel algorithm for sorting n elements evenly distributed over $2^d = p$ nodes of a d dimensional hypercube is given. The algorithm ensures that the nodes always receive equal number of elements (n/p) at the end, regardless of the skew in data distribution.

I. Introduction

This paper addresses the problem of sorting n elements evenly distributed over $2^d = p$ nodes of a d dimensional hypercube, where n >> p. The n elements are defined as sorted whenever a global order is obtained such that for $p-1 \ge i > j \ge 0$ any element in node i is greater than any element in node j, and within each node n/p elements are sorted among themselves. Main contributions presented in this paper are:

- 1. An enumeration sorting algorithm which ensures that the nodes always receive equal number of elements (n/p) at the end, regardless of the skew in data distribution. Running time of the algorithm is $O((n \log n)/p + p \log^2 n)$ for uniform data distribution.
- 2. A parallel selection algorithm which determines the p-1 partitioning keys used in sorting in $O(p \log^2 n)$ time. Best known previous result on selection [1] would yield

a time complexity of $O(p \log^2 p \log(n/p))$ for the same problem.

3. A communication algorithm used in sorting which eliminates the store-and-forward overhead by making use of the distance-d communication capability of the iPSC/2 hypercube system. This algorithm is substantially faster than the store-and-forward scheme.

Implementation results show that the sorting algorithm based on Items 2 and 3 above performs better than the *hyperquicksort* algorithm for large n [2].

Parallel sorting algorithms for distributed memory hypercube multiprocessors were previously given in [3, 2, 4, 5, 1, 6]. The enumeration sorting algorithms given in [2, 5, 4] do not address the problem of distributing data equally across the nodes; nodes do not necessarily finish the sort with n/p elements each, but depending on the skew and initial ordering of data, some nodes may end up with more than n/p elements. This may lead to two problems: 1) load imbalance during the sort, since some nodes have to process more than n/p elements, 2) insufficient amount of memory to complete the sort in some nodes. For example, hyperquicksort which is commonly known as the fastest practical sorting algorithm performs poorly in some cases such that nearly all n elements, instead of n/p end up in one node, limiting both the speedup and the useful range for n [2].

In the sorting algorithm described here elements are evenly distributed across the p nodes such that from start to finish each node processes n/p elements exact. This is accomplished by using the balanced partition keys for redistributing data among the nodes, contrary to other algorithms which select the partition keys either randomly or by sampling the elements. The balanced partition keys can be determined in the following manner: Let $L[1 \cdots n]$ be the final sorted list, the result of the sort. The p-1 elements L[kn/p] (k = 1, ..., p-1) correspond to the balanced partition keys, since any two keys L[kn/p]and L[(k+1)n/p] have exactly n/p elements between them in the final sorted list $L[1 \cdots n]$. If p-1 partitioning keys are available, every node can send all elements greater than or equal to L[kn/p] and smaller than L[(k+1)n/p] to node k, so that the elements are globally ordered and the total number of elements received by node **k** is exactly n/p. Therefore, main steps of the sorting algorithm can be given as follows:

- 1. Quicksort: Each node independently quicksorts the n/p elements initially residing in its memory to form a sorted list $A[0 \cdots n/p-1]$.
- 2. Select Partitioning Keys: Nodes run a parallel selection algorithm to determine the p-1 partitioning keys L[kn/p] ($k=1,\ldots,p-1$). This algorithm described in Section III runs in $O(p\log^2 n)$ average time.
- 3. Global Exchange: Each node finds the insertion point of the p-1 partitioning keys in its list $A[0\cdots n/p-1]$. This will partition the list into p segments, in general. The segment between the insertion points of L[kn/p] and L[(k+1)n/p] is sent to node k $(k=1,\cdots,p-2)$. Similarly, the segment below the insertion point of L[n/p] is sent to node 0, and the segment above the insertion point of L[(p-1)n/p] is sent to node p-1.
- 4. Binary Tree Merge: Each node has now received p-1 sorted segments from other

nodes and one from itself. Each node forms a single sorted list out of these p segments in $O((n \log p)/p)$ time using binary tree merge.

5. End of Algorithm.

Note that the time complexity of Steps 1, 2, and 4 add up to $O((n \log n)/p + p \log^2 n)$. Time complexity of Step 3 depends on global data distribution. For uniform distribution, it is O(n/p). In the next section we describe Step 3 in more detail. The parallel selection algorithm and implementation results are presented in Sections III and IV, respectively.

II. Global Exchange

Let A_{ℓ}^{i} $(\ell = 0, 1, \dots, p-1)$ denote the p sorted segments in node i induced by the balanced partition keys. In the global exchange step of the sort, segments are exchanged among the nodes such that each node i sends its segment A_i^* to node ℓ . These exchanges must be ordered such that segments do not collide and block each other on the hypercube links. We perform this task as in the following: In the iPSC/2 hypercube, each node is equipped with a direct connect module (DCM) which allows non-neighboring nodes to communicate directly [7], instead of using the store-andforward scheme [8]. A DCM is basically a (d+1)input, (d + 1) output crossbar switch. The d input-output pairs of the DCM are connected to the d neighbors of the node through hypercube links. The remaining input-output pair is connected to the internal bus of the node, hence to its memory. A DCM can be set up so that a message coming from one link can be immediately directed to another link, thereby eliminating the store-and-forward overhead. Our measurements on iPSC/2 indicate that this scheme is as fast as near-neighbor communication if all the links in the communication path are available. A fixed routing scheme called e-cube algorithm is used for routing the messages in iPSC/2 [7], where bit by bit logical exclusive-or of the source and destination node numbers gives the routing tag. Nonzero bit positions in the routing tag, when read from right to left, give the hypercube coordinate directions a message goes through. For example, if the routing tag is $r = (r_4r_3r_2r_1r_0) = (01011)$, message first goes in the coordinate direction 0, then direction 1, then direction 3, and finally arrives at its destination.

By making use of the DCMs and the e-cube scheme, the following global exchange algorithm coordinates the exchange of segments so that they are delivered to their destinations directly and that the communication paths used by the segments are always disjoint. Thus, segments never block each other on the network links. Each node distributively executes the following, where \oplus denotes an exclusive-or operation:

Let z be this node's id, and let $A_{k=0,\ldots,p-1}^z$ be the p-1 segments in node z for $k=1,\ldots,p-1$ send segment A_k^z to node $z\oplus k$ receive segment A_k^k from node $z\oplus k$ wait for receive and send to complete sync endfor

Processors wait at the sync instruction until it is executed by all of p them, which ensures that no processor gets ahead and occupy links used by other processors. It may be verified from Fig. 1 that the segments follow disjoint paths. An O(n) bottleneck can be present in the global exchange algorithm for some data distributions. For example, this will happen for a case where each node has to send its entire n/p size list to another node. The bottlenecks may be eliminated by removing the sync statement or by using different routing algorithms, but this is a subject of further analysis and will not be discussed here.

An interesting feature of our sorting algorithm is the ability to overlap communication and com-

putation in the global exchange and binary tree merge steps using asynchronous communication primitives; As soon as node z receives its first segment A_z^ℓ , it begins merging the pair of segments A_z^ℓ and A_z^r , while two more segments arrive to the node in parallel with the merge. Merging of segment pairs continue in this pipelined fashion until all of the segments are exchanged.

III. Selecting the Partition Keys

The partitioning problem here is selecting the n/p-th, 2n/p-th,...(p-1)n/p-th largest keys out of p sorted lists of size n/p each. A partitioning key, namely L[kn/p] $(k = 1, \dots, p-1)$, can be determined in a fashion similar to the ordinary binary search. An informal description of the algorithm will be given first: Assume, a key X is proposed as the partition key. Each node determines the number of elements smaller than X (referred to as local rank) in its sorted list $A[0 \cdots n/p-1]$. Local rank can be determined in $\log(n/p)$ comparisons using binary search. Summation of the p local ranks of X gives its global rank, hence its position in the final sorted list $L[1 \cdots n]$. If the global rank of X is greater(smaller) than kn/p, a new candidate smaller(greater) than X is proposed as the partition key, and the procedure is iterated until L[kn/p] is found. The number of iterations will be $\log_2 n$ on the average, if candidate partition keys are chosen properly. This idea is the basis of the partitioning algorithm presented next. We give only a sketch of the algorithm. Exact details can be found in [6]:

1. Initialize: Let $A[0 \cdots n/p - 1]$ be the sorted list of n/p elements in node i ($i = 0, \ldots, p-1$). Let the local variables min[k] and max[k] ($k = 1, \ldots, p-1$) be the pointers for the sorted list $A[0 \cdots n/p-1]$. During the iterations, the local search space for the k-th partition key L[kn/p] will always be between min[k] and max[k] such

that A[min[k]] < L[kn/p] < A[max[k]]. Set min[k] = -1, max[k] = n/p for $k = 1, \ldots, p-1$. Initially, each node will propose $A[kn/p^2]$ as a candidate for the k-th partition key L[kn/p] for $k = 1, \ldots, p-1$.

- 2. Transpose: Each node i (i = 0, ..., p 1) is now holding a candidate for the k-th partition key. All p candidates associated with k-th partition key are moved to node k for k = 1, ..., p 1. Node 0 gets only NIL values. This step can be completed in $\Theta(p \log p)$ time.
- 3. Select Median of the Candidates: Node k is now holding p candidates associated with the k-th partition key. Node k quicksorts these keys and then determines their median in $O(p \log p)$ time. (The median key will be referred to as the k-th candidate, meaning that it is the candidate for the k-th partition key.) Candidates other than the median are discarded. Node 0 is idle at this step.
- 4. Broadcast: Each node k (k = 1, ..., p-1) broadcasts the median key to rest of the nodes 0, 1, ..., p-1. This step takes $\Theta(p)$ time.
- 5. Local Rank Computation: Every node now has a copy of the p-1 candidates. Local rank R[k] of the k-th candidate in each node is determined by a binary search in $A[0 \cdots n/p-1]$. This step takes $O(p \log(n/p))$ time total.
- 6. Global Rank Computation: The p local ranks of the k-th candidate are summed in $\log_2 p$ communication and addition steps to give its global rank G[k] for $k = 1, \ldots, p-1$, resulting in $\Theta(p \log p)$ overall time for this step. If G[k] = kn/p, then k-th candidate is the balanced partition key L[kn/p].
- 7. Reduce the Search Space: If G[k] > kn/p, it is known that the k-th candidate is

greater than L[kn/p]. Therefore, each node decreases max[k] pointer to the candidate's insertion point in its list $A[0 \cdots n/p - 1]$. Likewise, if G[k] < kn/p, then each node increases min[k] pointer to the candidate's insertion point in its list $A[0 \cdots n/p - 1]$.

8. Propose New Candidates: For k = 1, ..., p - 1, each node proposes A[(max[k] + min[k])/2] as a candidate and the next iteration begins from Step 2 until all p-1 balanced partition keys are found.

9. End of Algorithm.

Each iteration of the algorithm takes $O(p \log n)$ time, and the number of iterations is $\log_2 n$ on the average, giving an average time complexity of $O(p \log^2 n)$. Note that the previous result in [1] yields $O(p \log^2 p \log(n/p))$ time complexity for the same problem.

The transpose operation in Step 2 is a general hypercube algorithm for distributing p values in every node to the rest of the nodes in in $\Theta(p \log p)$ time. Each of the p values in any given node is addressed to a different node. Let val in tuple $\langle val, dst, src \rangle$ denote the value to be sent from node src to node dst. In any given node z, there are p tuples $\langle val_z^z, j, z \rangle$ $(j = 0, 1, \ldots, p - 1)$ initially. Upon completion of the algorithm, node z receives the p tuples $\langle val_z^j, z, j \rangle$ $(j = 0, 1, \ldots, p - 1)$ addressed to it from other nodes. For example, assume that the following values are present on a 4 node system initially:

<u>k</u>	Node 0	<u>Node 1</u>	Node 2	Node 3
0	NIL	NIL	NIL	NIL
1	12	46	23	19
2	28	3	35	37
3	8	57	18	66

After transpose, contents of the nodes change as follows:

Node 0 Node 1 Node 2 Node 3

NIL	12	28	8
NIL	46	3	57
NIL	23	3 5	18
NIL	19	37	66

The transpose algorithm can be described as in the following: Let $z = (z_{d-1} \cdots z_0)$ be the binary representation of the node z's id, and $T[0 \cdots p-1]$ denote the list of p tuples $\langle val_j^z, j, z \rangle$ residing in node z.

for
$$m = 0, 1, ..., d - 1$$

- 1. split T into two lists B and B' such that B contains tuples whose j field agree with $(z_{d-1}\cdots z_0)$ in m-th bit position, and B' contains tuples which do not,
- 2. send B' to node $(z_{d-1}\cdots \overline{z_m}\cdots z_0)$ on coordinate m,
- 3. receive C from node $(z_{d-1}\cdots \overline{z_m}\cdots z_0)$ on coordinate m
- 4. $T \leftarrow B \cup C$

IV. Experimental Results and Conclusions

The parallel sorting algorithm was implemented for sorting 32 bit integers on an 8 node 386 processor based iPSC/2 hypercube multiprocessor. The hyperquicksort algorithm was also implemented for comparison [2]. The global exchange and the binary tree merge steps were implemented to allow communication and computation overlap as described earlier. The elements to be sorted are randomly generated in the nodes. Locally in each node, unsorted elements are uniformly distributed. To observe the effect of interprocessor communication during the global exchange step, three different global data distributions were used. The BALANCED distribution is adjusted such that the p-1 partition keys split each list of size n/p to p equal sized segments.

Hyperquicksort reaches its best performance at this distribution. Note also that the quicksort and merge steps of our algorithm and of hyperquicksort are supposed to take equal time for this distribution. Table 1, columns 3 and 4 show the sort times for hyperquicksort and our algorithm. respectively. Results show that as n gets large, our algorithm sorts faster (faster cases are indicated by *). Thus, just by the virtue of the global exchange algorithm described in Section II, better results were obtained for large n. The column Q indicates the quicksort and P indicates the partition time of our algorithm, and M indicates the global exchange and merge time of our algorithm. Number of iterations made by the partitioning algorithm is indicated in the last column. Table 1 shows that the number of iterations is approximately log₂ n for the BALANCED distribution case.

The WORST distribution is adjusted to induce an O(n) bottleneck in the global exchange step of our algorithm. All of the n/p elements initially in node i (i = 0, ..., p-1) have to move to node $i+1 \pmod{p}$ after sorting. Thus, in each node i, segment A_i^l has a size n/p if $l = i+1 \pmod{p}$, and has a size 0 if $l \neq i+1 \pmod{p}$. Since, exchange of segments are serialized with the sync instruction, it will take O(n) time to complete the global exchange step for this distribution. Table 2 shows that for large n hyperquicksort is slower. This was due to load imbalance in the merge step of hyperquicksort; some nodes had to merge more than n/p elements, while every node finished the sort exactly with n/p elements in our algorithm.

In the BEST distribution initially all elements in node i are greater than all elements in node j, if i > j. Since data is already globally ordered, exchange of segments during the global exchange step is eliminated. Table 3 shows that for large n our algorithm again performs better due to its smaller communication overhead, and due to load imbalance in the merge step of hyperquicksort. For example, for the case of d = 3 and n = 64,000 (not shown in the table), one

node finished the sort with 15,000 elements and one other with 1,000 elements in hyperquicksort, whereas every node finished the sort exactly with 8,000 elements in our algorithm.

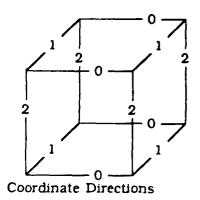
Results show that the sorting algorithm presented in this paper obtains very competitive speedups, and it has the advantage of equally distributing data over the hypercube. Main weakness of the algorithm is the relatively high partitioning overhead for small n. However, in practice the criteria $\epsilon > |kn/p - G[k]|$ can be used for terminating the iterations earlier, resulting in partitions of size $n/p \pm 2\epsilon$ at worst.

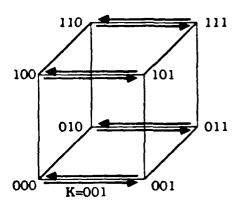
REFERENCES

- [1] C. Plaxton, "Load balancing, selection and sorting on the hypercube," in *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pp. 64-73, 1989.
- [2] B. Wagar, "Hyperquicksort," in Hypercube Multiprocessors 1987, SIAM Press., Phila., pp. 292-299, 1987.
- [3] S. L. Johnsson, "Combining parallel and sequential sorting on a boolean n-cube," in Int'l Conf. on Parallel Processing, pp. 444-448, 1984.
- [4] S. R. Seidel and W. L. George, "Binsorting on hypercubes with d-port communication," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 1455-1461, January 1988.
- [5] G.Fox, M.Johnson, G.Lyzenga, S. J.Salmon, and D.Walker, Solving Problems on Concurrent Processors, Vol I. NJ: Prentice-Hall, 1988.
- [6] B. Abali, Sorting Algorithms for Hypercube Multiprocessors. PhD thesis, The Ohio State University, Columbus, Ohio, 1989.
- [7] S. Nugent, "The iPSC/2 direct-connect communications technology," in *Proceedings of*

the Third Conference on Hypercube Concurrent Computers and Applications, pp. 51-60, January 1988.

[8] C. L. Seitz, "The cosmic cube," Comm. ACM. vol. 28, pp. 22-33, Jan. 1985.





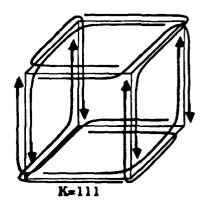


Figure 1: Some steps of the Global Exchange on a 3-cube

Table 1: For the BALANCED distribution case, execution times of Balanced-Partition Sort and Hyperquicksort (msec.)

d	nx10 ³	Нур.	Total	Q	P	M	I
1	4	77	104	56	31	17	13
1	8	158	178	117	32	29	13
1	16	325	338	246	33	59	14
1	3 0	639	* 639	495	37	107	15
1_1	6 0	1320	*1288	1034	38	216	16
2	4	51	105	27	58	2 0	13
2	8	98	142	55	54	33	12
2	16	196	236	118	59	59	13
2	3 0	374	409	234	68	107	15
2	6 0	771	778	495	74	209	16
2	100	1 33 0	*1294	871	78	345	17
2	12 0	1583	*1524	1034	78	412	17
3	4	38	132	13	95	24	13
3	8	64	168	25	109	34	15
3	16	120	224	54	117	53	16
3	3 0	222	317	109	118	90	16
3	6 0	448	524	234	125	165	17
3	100	753	792	401	126	265	17
3	12 0	917	943	496	133	314	18
3	15 0	1152	*1150	627	133	39 0	18
3	2 00		1522	871	136	515	18
3	24 0		1791	1034	143	614	19
3	3 00		2252	1346	141	765	19

Table 2: For the WORST distribution case, execution times of Balanced-Partition Sort and Hyperquicksort (msec.)

d	nx10 ³	Нур.	Total	Q	P	M	I
1	4	78	119	56	54	9	23
1	8	159	188	118	55	15	23
1	16	327	335	246	60	29	25
1	3 0	645	*613	495	65	53	27
1	6 0	1315	*1207	1034	69	104	29
2	4	54	147	26	98	23	22
2	8	102	194	55	98	41	22
2	16	207	302	117	107	78	24
2	3 0	397	493	234	117	142	26
2	6 0	803	900	495	126	279	28

Table 2: For the WORST distribution case, execution times of Balanced-Partition Sort and Hyperquicksort (msec.)

3	4	44	185	13	149	23	21
3	8	78	211	26	150	35	21
3	16	149	279	55	165	59	23
3	3 0	275	388	109	179	100	25
3	6 0	562	617	234	194	189	27
3	100	941	*916	400	209	3 07	29
3	120	1139	*1069	495	208	366	29
3	150	1431	*1308	627	226	455	31

Table 3: For the *BEST* distribution case, execution times of Balanced-Partition Sort and Hyperquicksort (msec.)

d	nx10 ³	Нур.	Total	Q	P	M	I
1	4	75	115	56	54	5	23
1	8	155	180	117	55	8	23
1	16	318	322	246	60	16	25
1	3 0	631	*589	496	64	29	27
1	6 0	1 3 02	*1161	1033	7 0	58	29
2	4	55	131	26	97	8	22
2	8	107	163	55	97	11	22
2	16	216	243	118	106	19	24
2	3 0	414	*381	234	115	32	26
2	6 0	852	*682	495	125	62	28
2	100	1462	*1106	871	135	100	3 0
2	120	1723	*1288	1033	135	120	3 0
3	8	79	190	26	149	15	21
3	16	15 0	241	55	165	21	23
3	3 0	279	318	109	178	31	25
3	6 0	566	*480	233	194	53	27
3	100	952	*694	400	210	84	29
3	120	1155	*803	495	210	98	29
3	150	1449	*969	626	224	119	31
3	200		1251	871	224	156	31
3	240		1445	1034	224	187	31
3	30 0		1816	1346	240	23 0	33

Parallel Sorting on the Hypercube Concurrent Processor

Tillie Tang

Mission Profile and Sequencing Section
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Dr Pasadena, CA 91109

Abstract

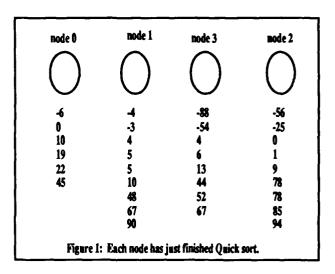
This paper describes a generalized version of a previously published parallel sort algorithm, "parallel shell merge".[3] This version was implemented on the JPL/Caltech Mark III hypercube concurrent processor. Each node starts out with a sublist of items to be sorted first internally, and then among the other nodes. Parallel shell merge is an algorithm used to produce a whole sorted list across the hypercube once each sublist is sorted internally. This version is general in the sense that it allows sublists of very different sizes to be sorted as well as being able to handle balanced sublists. This general version performs quite well when it is used to sort balanced loads of data; however, there are some efficiency losses due to the generalization, but they are acceptable.

I. Background

A parallel prototype of SEQGEN, the software that verifies and expands high-level activities into low-level command sequences for JPL flight projects, is in the process of development on the JPL/Caltech Mark III hypercube concurrent processor.[1][2] The purpose of this prototype is to show that the process of generating commands and sending them to the spacecraft, generally called "uplink", can be greatly sped up utilizing parallel computers. Since SEQGEN spends a significant portion of its time sorting commands in time order, a parallel sort algorithm is necessary for the best possible parallel speed up.

II. Parallel Sort

The parallel sort works as follows. Each node of the hypercube starts out containing a sublist of commands in which there are time fields to be sorted. Each sublist can be of any arbitrary size. The algorithm to be described is a generalization of shell merge.[3] The algorithm's prerequisite is that the sublist on each node is sorted; therefore, the sequential version of quick sort is applied to each sublist independently using the median-of-three method to select a pivot point where partition starts. Once each sublist is sorted independently by quick sort as



illustrated in Figure 1, shell merge begins. These sublists lying across the hypercube make up the complete list. Shell merge manipulates the resulting sublists and sorts them in ascending order across the hypercube. In other words, shell merge compares and exchanges the items of each pair of the resulting sublists such that the node on the left would have the lower-valued items while the higher-valued items migrate to the node on the right. The node order is defined such that they reflect a map of the hypercube topology onto a one-dimensional array, with the lowest order nodes on the left. For example, in Figure 2 the case of an 8-node cube is shown to have the following order: 0, 1, 3, 2, 6, 7, 5, 4. Thus, if node 0 and node 1 were to perform compare and exchange as mentioned above, node 0 would be the node on the left and node 1 is the node on the right as depicted in Figure 3.

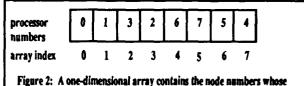
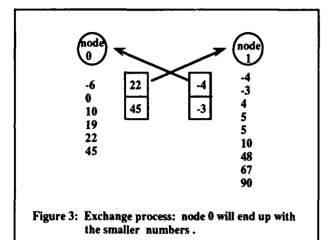
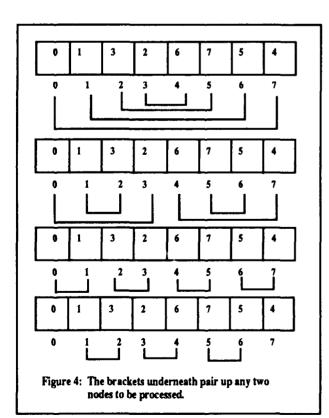


Figure 2: A one-dimensional array contains the node numbers whose order is mapped to the hypercube topology.

Figure 4 illustrates the scheme used in shell merge to determine which pair of nodes are to do compare-exchanges with one another. The first part of the algorithm accomplishes most of the sorting efforts in d (dimension of cube) steps. The algorithm makes large jumps to get the items to their destinations resulting in an almost-sorted list. The remaining part of the algorithm is the mop-up stage where an algorithm resembling bubble sort is implemented to complete the sort.[3]





The generalization of the parallel shell merge algorithm described in this paper handles the case where the sublists are of very different sizes, e.g. node 0 has a list with 30 elements and node 1 has a list with 5 elements. Since we are using a synchronous, "crystalline" environment, it is necessary to avoid deadlocks in communication between two nodes, as the node with shorter sublist is done and well on its way to other tasks while the one with the longer sublist waits indefinitely for the other node to talk back to it. If the data is arranged in such a way that regardless of the different list lengths, the node with the longer sublist does not need to communicate with the other node after the other node is done, then the algorithm works fine. Although cases like this where each node has a very different sublist size than its neighbor node do not make the best use of parallelism, an algorithm should still be general enough to handle these cases, especially to fulfill our particular application.

Shell merge is able to sort sublists of very different sizes. This is accomplished by detecting when the node with the shorter sublist has just finished comparing and exchanging data with its partner whose sublist is longer and by stopping the node with the longer sublist from asking for data from the node with the shorter sublist. When the node with the shorter sublist has traversed all of its elements, the sorting between the two nodes are completed; thus, no further communications should be attempted by the node with the longer sublist. To detect when the two nodes should stop communicating with one another (when the node with the shorter sublist is done), a flag called "end_of_list" is set. This flag is set as soon as the number of items read from the other node is found to be greater than or equal to the length of the smaller of the two sublists. Since both nodes are always checking for the stopping point, once either node's "end_of_list" flag is set, the communication ends.

When shell merge is completed, node 0 is supposed to have the sublist with the lowest items while the node at position (n-1) of the array (node 4, in our 8-node example) should contain the sublist with the highest items since node 0 is the first node in the one-dimensional array of processor numbers and the (n-1)st node is the last node in the array (where n = number of nodes). During the mop-up stage, if one allows node 0 and the (n-1)st node to communicate, the higher node would have the low-valued sublist while the lower node would have the high-valued sublist, and hence the sort would never complete.

To avoid this, a nonperiodic boundary bookkeeping

scheme was developed: when node 0 is about to communicate with its neighbor to the left, the channel mask of these two nodes must be determined (where the channel mask is a decimal number obtained by converting the binary number whose individual bits represent the communication channels of the cube). However, notice that node 0 does not have a neighbor to its left; node 0 is the leftmost node in the array. Hence, when node 0 attempts to determine its channel mask with its left neighbor, the channel mask is set equal to 0 to prevent it from trying to communicate with the (n-1)st node. The same is done when the (n-1)st node attempts to communicate with its neighbor to the right since the (n-1)st node is the rightmost node. If this special case is not handled as described above, node 0 and the (n-1)st node would communicate with one another since the one-dimensional array is treated as a circular array by the Crystalline Operating System routine "gridchan" where node 0 is adjacent to the (n-1)st node.

Figure 5 illustrates how the one-dimensional array can be thought of as circular. Note that in order for the (n-1)st node to be seen as node 0's left neighbor or for node 0 to be seen as the (n-1)st node's right neighbor, one must look at the circle in Figure 5 always facing the center of the circle while standing at the position of each array index. In order for the list to be sorted in ascending order from left to right along the array, the node on the left must keep the lowest-valued items to itself while sending away the highest-valued items to the node on its right. Therefore, if node 0 and the (n-1)st node were to communicate with each other, node 0 would treat the (n-1)st node as its left neighbor and would try to keep the highest-valued items to itself while sending away the lowest-valued items to the (n-1)st node. This obviously contradicts the intended algorithm, and the same kind of contradition results when the (n-1)st node is to communicate with node 0 treating node 0 as its right neighbor. Because of this circular action, an infinite loop would result and the list would never be sorted.

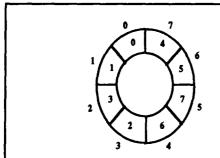
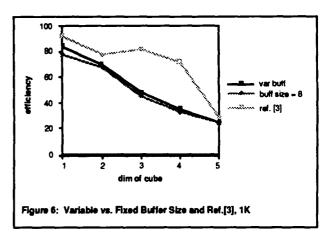
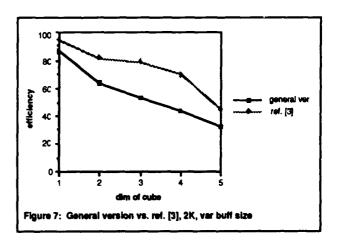
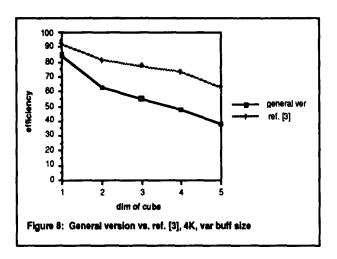


Figure 5: The numbers inside the slots are processor numbers; the ones outside are array indices.

In order for a node on the left hand side, e.g. node 3, to have a sublist of lower items than those of its right-hand-side neighbor, e.g. node 2 on a cube of 8 nodes, node 3 must send all of its highest items to node 2 while node 2 must send all of its lowest items to node 3. When sending items between two nodes for this purpose, only portions of each sublist are sent at a time. The bigger the portions which are sent at once, the less time is wasted while a node sits idle waiting for the other node to send some more items. In addition, the smaller these portions are, the more communication calls are required. However, if the whole sublist is sent all at once, then there is the risk of unnecessary transfer of data, depending on the nature of the actual data to be sorted. Therefore, the size of the buffer is allowed to vary from case to case depending on the size of the smaller sublist of the two nodes communicating. Empirically, 40% of the length of the smaller sublist is a reasonable buffer size. This not only avoids hard coding, but it also improves the speed as shown below in Figure 6 for a case with 1024 total items to be sorted.







III. Conclusions and Results

The parallel sort described above is a generalized version that works regardless of the imbalanced sizes of the sublists to be sorted. Although imbalanced sublists would not make use of parallelism to its fullest potential, an algorithm should still be general enough to withstand the worst input. As shown in Figures 6, 7, and 8, this general version performs quite well when it is used to sort balanced loads of data (relatively constant sizes of sublists). There are some efficiency losses due to its generalization. It can be seen from Figures 6, 7, and 8 that the cost of having a more general algorithm is about a factor of 10% to 50% loss in efficiency from the same-list-size algorithm, depending upon problem size. Interestingly, though, note that the loss is not monotonic with number of nodes (the general algorithm has a more linear fall-off).

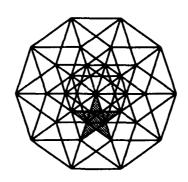
IV. Acknowledgements

The work described above was completed with invaluable assistance and guidance from Joan Horvath, Edith Huang, Nooshin Meshkaty, Barbara Zimmerman, and Bob Cole. The author would also like to thank the JPL Hypercube Project and Dave Curkendall for the hypercube time made available for this application. The work described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

V. References

[1] Horvath, J. C., and Cole, R. C. "Spacecraft Sequencing on the Hypercube Concurrent Processor", Presented at the Fourth Conference on Hypercube Concurrent Computers and Applications, Monterey, CA March 1989.

- [2] Horvath, J. C., Tang, T., Perry, L. P., Cole, R. C., Olster, D. B., and Zipse, J. E., "Hypercubes for Critical Space Flight Command Operations." To be presented at DMCC5, Charleston, SC April 1990.
- [3] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D. <u>Solving Problems on Concurrent Processors</u>. Prentice Hall, Englewood Cliffs, New Jersey 1988.



The Fifth Distributed Memory Computing Conference

10: Mathematical Methods

Parallel Methods for Solving Polynomial Problems on Distributed Memory Multicomputers

Xiaodong Zhang *
Division of Mathematics and Computer Science
The University of Texas at San Antonio
San Antonio, Texas 78285-0664

Hao Lu
Department of Mathematics
Xian Jiaotong University
Shanxi, People's Republic of China

Abstract

We give a group of parallel methods for solving polynomial related problems and their implementations on a distributed memory multicomputer. These problems are 1. the evaluation of polynomials, 2. the multiplication of polynomials, 3. the division of polynomials, and 4. the interpolation of polynomials. Mathematical analyses are given for exploiting the parallelisms of these operations. The related parallel methods supporting the solutions of these polynomial problems, such as FFT, Toeplitz linear systems and others are also discussed. We present some experimental results of these parallel methods on the Intel hypercube.

1 Introduction

Polynomials are one of the most useful and well known classes of functions in various applications. A polynomial function is defined as

$$P_n(x) = a_0 + a_1 x + \dots + a_n x^n \tag{1}$$

where n is an nonnegative integer and a_0 , ..., a_n are real constants. When the degree of a polynomial function, n is very large, the computations for polynomial problems, such as the interpolation of polynomials, multiplication and division of polynomials are intensively required. In addition, polynomials represent an important class of expressions in algebraic manipulation in symbolic computation. The basic polynomial arithmetic operations such as multiplications and divisions spend huge execution times in computer algebra processing (see e.g. Ponder[1988], Siebert-Roch and Muller[1989]). Thus, such computation problems are good candidates for parallel computers both numerically and symbolically.

We give a group of parallel methods for solving the polynomial related problems, and their implementations on a distributed memory multicomputer. These problems are: 1. the evaluation of polynomials, 2. the multiplication of polynomials, 3. the division of polynomials, and 4. the interpolation of polynomials. Parallel evaluation method of

polynomials based on the Horner's rule is discussed in section 2. The experimental results on the Intel hypercube are also presented. The parallelism of the polynomial multiplication is exploited by transferring the problem to a set of special FFT series functions, on which the operations can be perfectly distributed among different processors. Section 3 gives the mathematical analyses and parallel method of the polynomial multiplication. The polynomial division problem is solved based on parallel solutions for Toeplitz triangular linear systems and the parallel polynomial multiplication, and is discussed in section 4. Section 5 addresses a parallel method for the Lagrange piecewise cubic polynomial interpolation. Finally, we give a summary and future work in the last section.

2 Parallel evaluation of polynomials

The evaluation of a polynomial function is a basic operation in polynomial problems:

$$P_n(x_0) = a_0 + a_1 x_0 + ..., a_{n-1} x_t^{n-1} + a_n x_0^n \qquad (2)$$

where $a_0, a_1, ..., a_n$ and the indeterminate x_0 are the input variables, and $P_n(x_0)$ is the output solution variable. A straight forward parallel method for (2) is to partition the evaluation operations into a binary tree structure so that the suboperations can be distributed among the processors (see e.g. Siva and Murthy [1989]). The drawbacks of this method are that the number of multiplications required is not minimized, and large amount of communication are involved among different layers of the tree processors in the process of evaluation.

Horner's rule, which evaluate a polynomial by the scheme

$$P_n(x_0) = (\dots ((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_1)x + a_0$$

requires exactly n multiplications, and is the optimal method to evaluate a polynomial in terms of minimizing the operations. The Horner's method may be easily described as a sequential recurrence:

$$\begin{cases}
b_n = a_n \\
b_{i-1} = b_i x_0 + a_{i-1} & i = n, n-1, ..., 1
\end{cases}$$
(3)

^{*}This author is supported in part by the University of Texas at San Antonio Faculty Research Award.

Based on (3), we develop a parallel Horner's method to evaluate a polynomial. Let $a_{1-j} = 0$, j = 2, ..., p, where p is number of processors used to evaluate a polynomial.

Do j = 1 to p in parallel begin

$$b_{n-j+1} = a_{n-j+1}$$
for $i = n - (p+j-1)$ to $p-j+1$ step $-p$

$$b_i = x_0^p b_{i+p} + a_i;$$

$$b_{1-j} = x_0^{p-j+1} b_{p-j+1} + a_{1-j}$$

 \mathbf{end}

$$P_n(x_0) = \sum_{i=1}^p b_{1-i}$$

The evaluation operations distributed among p processors are independent except the last step to collect and add all sub-solutions among the p processors. Thus the speedup can be determined by

$$S_p = \frac{n}{n/p + p - 1 + t_c} \tag{4}$$

where n is the number of operations (in time unit) for an n^{th} degree polynomial to be evaluated on a sequential machine, and $n/p+p-1+t_c$ is the number of operations (in time unit) plus the communication time t_c to collect the subsolutions among the p processors for evaluating the same polynomial on a multicomputer with p processors. The communication time t_c is trivial comparing with other operations since the operation to add all subsolutions among the p processors can be done through a global tree: the subsolutions are accumulated from the leaf level, then send to the host. (see Moler [1986]) This method minimizes the communication times to $log_2(p)$ instead of p in a sequential message transfer. Thus, when $n/p >> p-1+t_c$, we may obtain a close linear speedup $S_p \approx p$. However, the worest case is when p = n, the method become a sequential one and speedup $S_p \leq 1$. In addition, this parallel algorithm only applies to the situation when $n \mod p = 0$.

The parallel Horner's method has been implemented on a Intel hypercube multicomputer. Initially, each processor is distributed following coefficients and variables:

n: the degree of the polynomial;

p: number of processors used;

j: the processor index;

 x_0 : the indeterminate variable;

 a_i , i = p + j - 1, ..., p - j + 1: the coefficients for the j^{th} processor;

and coefficient an-j+1.

A polynomial of n=128 is evaluated on different number of processors on the hypercube. The speedup's with different number of processors are plotted in Figure 1.

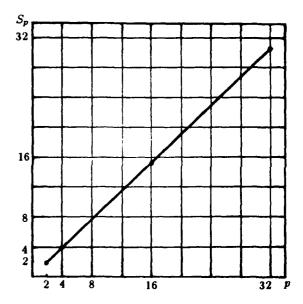


Figure 1: Performance of the parallel Horner's method on hypercube

3 Parallel method for the multiplication of polynomials

Let F(x) and G(x) be two n-1th degree polynomial functions.

$$F(x) = a_0 + a_1 x + \dots + a_{n-1} x^{n-1}$$
 (5)

and

$$G(x) = b_0 + b_1 x + ... + b_{n-1} x^{n-1}$$
 (6)

the multiplication problem is then defined as

$$F(x) \times G(x) = d_0 + d_1 x + ... + d_{2n-2} x^{2n-2}$$
 (7)

where

$$d_k = \sum_{\substack{i+1=k \ i}} a_i b_i$$
 $k = 0, 1, ..., 2n-2$.

Given two polynomials p_1 and p_2 , the following relations show how the degree and size of polynomials change under the operations of multiplication (see e.g. Fateman [1974]):

$$degree(p_1 \times p_2) = degree(p_1) + degree(p_2)$$

$$size(p_1 \times p_2) \leq size(p_1) \times size(p_2)$$
.

The coefficients in polynomials (5) and (6) may be defined as following two sequences respectively:

$$A(k) = \begin{cases} a_k & 0 \le k \le n-1 \\ 0 & n-1 < k \le 2n-1 \end{cases}$$

$$B(k) = \begin{cases} b_k & 0 \le k \le n-1 \\ 0 & n-1 < k \le 2n-1. \end{cases}$$

where A(k) and B(k) functions are periodic of period 2n such that

$$A(k+2nl) = A(k)$$

and

$$B(k+2nl) = B(k)$$

where k = 0, 1, ..., 2n - 1, and l is an integer. Then

$$D(i) = \sum_{k=0}^{2n-1} A(k)B(i-k) \quad i = 0, \dots, 2n-1$$
 (8)

The coefficients d_1 , i = 0, ..., 2n - 2 of the multiplication in (7) can be determined from (8):

$$d_i = D(i), \quad i = 0, 1, ..., 2n - 2.$$
 (9)

Thus, the major work for the multiplication is to compute the periodical function of D(i) for i = 0, ..., 2n - 1 in (8). We decompose the computation for D(i) in following steps with the aid of the discrete Fourier transform series:

(i)
$$x(j) = \sum_{k=0}^{2n-1} A(k)w^{jk}$$
 $j = 0, ..., 2n-1$

(ii)
$$y(j) = \sum_{k=0}^{2n-1} B(k)w^{jk}$$
 $j = 0, ..., 2n-1$

(iii)
$$z(j) = x(j)y(j)$$
 $j = 0, ..., 2n-1$

and finally,

(iv)
$$D(k) = \frac{1}{2n} \sum_{j=0}^{2n-1} z(k) w^{-jk}$$
 $k = 0, ..., 2n-1$

The value w is a complex number, $w = e^{-i2\pi/2n}$, where $i = \sqrt{-1}$, which is also called $2n^{th}$ primitive root of 1. In other words, $w^{2n} = 1$, and every other $2n^{th}$ root of 1 can be represented as some power of w.

The Operations in (iii) can be done perfectly in parallel. Notice that the functions in (i), (ii) and (iv) are similar to the ones in standard discrete Fourier transform and its inverse (see e.g. Aho, Hopcroft and Ullman [1974]):

$$f(j) = \frac{1}{n} \sum_{k=0}^{n-1} x(k) w^{jk} \quad j = 0, ..., n-1$$
 (10)

and

$$x(k) = \sum_{j=0}^{n-1} f(j)w^{-jk} \quad k = 0, ..., n-1$$
 (11)

where w is an n^{th} primitive root of 1, and f(j) is a real function for j = 0, ..., n - 1. Thus, the transforms defined for computing polynomial multiplication are special cases of the standard discrete Fourier transform.

The major operations in (i), (ii), and (iv) can be easily written in the form of matrix vector multiplication:

$$t = Wq (12)$$

where $t = (t_0, t_1, ..., t_{2n-2}, t_{2n-1})^T$,

$$W = \begin{pmatrix} w_{2n}^{0} & w_{2n}^{0} & w_{2n}^{0} & \dots & w_{2n}^{0} \\ w_{2n}^{0} & w_{2n}^{1} & w_{2n}^{2} & \dots & w_{2n}^{2n-1} \\ \vdots & \vdots & & \vdots & & \vdots \\ w_{2n}^{0} & w_{2n}^{2n-1} & w_{2n}^{2(2n-1)} & \dots & w_{2n}^{(2n-1)(2n-1)} \end{pmatrix}$$

and $q = (q_0, ..., q_{n-1}, 0, ..., 0)^T$. The *t* vector represents the *x*, *y* or *D* vector in (*i*), (*ii*) or (*iv*). The *q* vector represents the two sequences *A* or *B* of period 2n in (8). The complexity of the multiplication in (12) is $O(4n^2)$.

The Fast Fourier Transform (FFT) developed by Cooly and Tukey [1965] has been widely used for computing the standard discrete Fourier transform of (10) and (11). The complexity of the FFT reduces to $O(nlog_2n)$. This improvement comes from a reordering of data by taking advantage of the fact

$$w_n^{jk} = w_n^{jk \mod n}$$

for j = 0, ..., n-1 and k = 0, ..., n-1. The FFT computation structure can also be easily processed in parallel with some control of synchronization and communication. The implementation and experiments of parallel FFT methods have been done on both distributed memory and shared memory multiprocessors (see e.g. Chamberlain [1986], Chan[1986], and Norton and Silberger[1988]).

We apply FFT to our special series functions in (i), (ii) and (iv) with slight modifications. Since the functions are periodic of period 2n, and the q vector in the multiplication form (12) is only half full, the complexity to compute (i), (ii) and (iv) reduces to $O(nlog_2n - n)$.

The parallel method detailed below fulfills the requirement as every processor involved immediately in the computation, and no arithmetic operation on the same data sequence terms is ever duplicated on any pair of processors. The best arrangement of the processors on a distributed memory multicomputer is a hypercube topology for this FFT type processing in terms of communication efficiency since the exchange of values are only required between neighbor processors.

Consider a multiplication of two 3rd degree polynomials, i.e. n = 4 in the polynomial defined in (5) and (6). Substitute n = 4 to the matrix vector multiplication

$$t = Wq$$

we have
$$t = (t_0, t_1, ..., t_7)^T$$

and $q = (q_0, q_1, q_2, q_3, 0, 0, 0, 0)^T$. Using the property of period 2n

$$w_{2n}^{jk} = w_{2n}^{jk \mod 2n}$$

for j = 0, ..., 2n - 1 and k = 0, ..., 2n - 1, the W matrix becomes

$$W = \begin{pmatrix} w_8^0 & w_8^0 & w_8^0 & \dots & w_8^0 \\ w_8^0 & w_8^1 & w_8^2 & \dots & w_8^7 \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ w_8^0 & w_8^7 & w_8^6 & \dots & w_8^1 \end{pmatrix}$$

The matrix vector multiplication for n = 4 may be decomposed to its parallel form:

$$t = W_4 W_2 q'$$

where W_4 is a 4 block diagonal matrix

W2 is a 2 block diagonal matrix

$$W_{2} = \begin{pmatrix} w_{8}^{0} & 0 & w_{8}^{0} & 0 & & & & & \\ 0 & w_{8}^{0} & 0 & w_{8}^{0} & & & & & & \\ w_{8}^{4} & 0 & w_{8}^{4} & 0 & & & & & & \\ 0 & w_{8}^{4} & 0 & w_{8}^{4} & & & & & & & \\ & & & & w_{8}^{2} & 0 & w_{8}^{2} & 0 & & & \\ & & & & 0 & w_{8}^{2} & 0 & w_{8}^{2} & 0 & & \\ & & & & w_{8}^{6} & 0 & w_{8}^{6} & 0 & & & \\ & & & & 0 & w_{8}^{6} & 0 & w_{8}^{6} & 0 & & & \end{pmatrix}$$

and $q' = (q_0, q_1, q_2, q_3, q_0, q_1, q_2, q_3)^T$. In general, the matrix vector multiplication of order 2n in (12) may be decomposed into the form of

$$t = W_{n} \dots W_{1}W_{2}q' \tag{13}$$

where W_i is the *i* block diagonal matrix, for $i = 2, 2^2, 2^3, ..., n$, and $q' = (q_0, ..., q_{n-1}, q_0, ..., q_{n-1})^T$.

The formula (13) can be performed perfectly in parallel on a p processor multicomputer as long as each processor keeps a copy of variables $q_1, ..., q_{n-1}$, and matrix W_i , for $i = 2, 2^2, ..., n$ are distributed by rows to the p processors. Figure 2 gives the operation dependency graph in each processor for the example of n = 4 on a 4 processor multicomputer.

Figure 2: Operation dependency graph for a polynomial multiplication of order n = 4 on a 4 processor multicomputer.

4 Parallel method for the division of polynomials

Given a 2n degree polynomial A(x), and a n degree polynomial B(x),

$$A(x) = a_0 + a_1 x + \dots + a_{2n} x^{2n}$$

and

$$B(x) = b_0 + b_1 x + ... + b_n x^n$$

the division problem is then defined as:

$$A(x) = B(x)Q(x) + R(x)$$
 (14)

where Q(x) is the quotient polynomial of $A(x) \div B(x)$, and R(x) is the remainder polynomial.

We can show that the coefficients of the polynomial Q(x) with variables of $(x^n, x^{n-1}, ..., x^0)$ are identical to the coefficients of the polynomial $D(x) \times K(x)$ of the variable set $(x^{2n}, x^{2n-1}, ..., x^n)$, where D(x) is the quotient polynomial for

$$x^{2n} = B(x)D(x) + S(x)$$
 (15)

and K(x) is the polynomial formulated from A(x),

$$K(x) = a_n + a_{n+1}x + ..., a_{2n-1}x^{n-1} + a_{2n}x^n$$

(see e.g. You[1983]). Thus, the major task in the division of polynomials is to compute the coefficients of the polynomial D(x).

In (15), D(x) is defined as

$$D(x) = d_0 + d_1 + ..., d_{n-1}x^{n-1}d_nx^n.$$
 (16)

Substitute (16) to (15), and compare the coefficients on both sides of the polynomial, we get

$$BD = I \tag{17}$$

where

$$B = \begin{pmatrix} b_n & b_{n-1} & \dots & b_0 \\ & b_n & \dots & & \\ & & \dots & \ddots & \\ & & & \dots & b_{n-1} \\ & & & b_n \end{pmatrix},$$

is an upper triangular Toeplitz matrix formed from the given coefficients of polynomial B(x), and

$$D = \begin{pmatrix} d_n & d_{n-1} & . & . & b_0 \\ & d_n & . & . & . \\ & & . & . & . \\ & & & . & d_{n-1} \\ & & & & d_n \end{pmatrix},$$

is another upper triangular Toeplitz matrix formed from the unknown coefficients of polynomial D(x), and I is the identity matrix. From (17), the coefficients of d_i for i = 0, 1, ..., n may be determined by following triangular Toeplitz linear systems of equations:

$$\begin{pmatrix} b_{n} & b_{n-1} & \dots & b_{0} \\ & b_{n} & \dots & & & \\ & & & \ddots & & \\ & & & & b_{n-1} \\ & & & & b_{n} \end{pmatrix} \begin{pmatrix} d_{0} \\ & \\ & \\ & \\ d_{n-1} \\ d_{n} \end{pmatrix} = \begin{pmatrix} 0 \\ & \\ & \\ 0 \\ 1 \end{pmatrix}$$

Several parallel algorithms have been developed for solving the full and upper triangular Toeplitz linear system of equations. (see e.g. Bini[1984], Sai, Li and Xie[1986]). These algorithms are based on the recurrence methods for solving the Toeplitz linear systems. Parallelisms are exploited in the vector processing in each step of the recurrence. Since the computation of each step in the recurrence is dependent on the results of the previous step, and the sizes of the vector of each step increase as the recurrence steps increase, those algorithms would not be efficient to gain good speedups in a distributed memory multicomputer. Li and Coleman [1988] [1989] develop a group of parallel methods for solving general triangular linear system on distributed memory multicomputer. Based on the idea of Li-Coleman's methods, and the Toeplitz triangular system solution dependency graph in Figure 3, we give a parallel method for solving the Toeplitz triangular linear system (18). Assume the Toeplitz triangular matrix is distributed in a wrap fashion among the processors: column j is assigned to processor (j-1) mod p, where j=0,1,...,n and p is the number of processors used. Thus the columns distribution function is defined as

$$P(j) = (j-1) \bmod p$$

The initial conditions are to set following two vector variables Sum(), the partial sum results for the solution, and Psum() the partial sum results computing in parallel:

if
$$nodenum = P(n)$$
 then begin

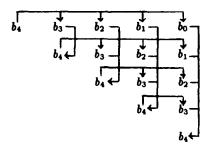


Figure 3: Toeplitz triangular system solution dependency graph

$$Sum(1) = 1;$$

 $Sum(i) = 0$ for $i = 2, ..., p;$
 $Psum(i) = 0$ for $i = 1, ..., n - p + 1;$
end
else
begin
 $sum(i) = 0$ for $i = 1, ..., p;$
 $Psum(i) = 0$ for $i = 1, ..., n - p + 1$
end.

The program on each processor:

```
for j = n downto 1 do begin

if nodenum = P(j) then
begin

receive Sum(i) for i = 1, ..., p-1 and j < n;
d_j = (Sum(1) + Psum(j))/b_n;

Sum(i) = Sum(i+1) - a_{n-1}d_j + Psum(j-i) for i = 1, ..., p-2;

Sum(p-1) = a_{n-p+1}d_j + Psum(j-p+1);

Send Sum(i) to node P(j-1)

for i = 1, ..., p-1 and j > 1;

Psum(i) = a_{n-j+1} for i = 1, ..., n-p
end
end.
```

The coefficient of quotient polynomial, Q(x) is determined by $K(x) \times D(x)$, in which, a parallel multiplication described in the last section is applied. Then the remainder polynomials R(x) is determined by

$$R(x) = A(x) - B(x)Q(x),$$

where $B(x) \times Q(x)$ uses the parallel multiplication method again.

5 Parallel method for Lagrange piecewise cubic interpolation

Given a set of n+1 pairs of real values, (x_i, y_i) for i=0,1,...n with distinct $x_i's$, there exists a unique polynomial

 $P_n(x)$ of degree n such that $P(x_i) = y_i$ for i = 0, 1, ..., n. This interpolating polynomial $P_n(x)$ can be written in the Lagrangian form

$$P_n = \sum_{i=0}^n y(x_i)l_i(x) \tag{19}$$

where

$$l_i(x) = \prod_{k=0}^{n} \frac{(x-x_k)}{(x_i-x_k)}$$

for $i \neq k$ and i = 0, 1, ..., n.

The Lagrange form of (19) may be decomposed as follows

$$P_n = \sum_{k=0}^{n} \hat{C}_i \prod_{k=0}^{n} (x - x_k)$$
 (20)

where

$$\hat{C}_i = \prod_{k=0}^n \frac{y_i}{(x_i - x_k)}$$

for $i \neq k$ and i = 0, 1, ..., n. The coefficients \hat{C}_i can be calculated in parallel perfectly on a distributed multicomputer. The paralle algorithm on p processors is in following form:

Do
$$j=1$$
 to p in parallel begin for $i=(j-1)[n/p]$ to $j[n/p]-1$ do begin $h=1;$ for $k=0$ to n and $(k\neq i)$ $h=h\frac{1}{x_i-x_k};$ $\hat{C}_i=hy(x_i)$ end end.

The polynomial evaluation on x_0 point based on the decomposed Lagrange form (20) can be done in parallel for most of the calculation except the sum operations at the end.

Do
$$j=1$$
 to p in parallel
begin
 $S_j=0$;
for $i=(j-1)[n/p]$ to $j[n/p]$ do
begin
 $h=1$;
for $k=1$ to n and $(k\neq i)$
 $h=h(x_0-x_k)$;
 $S_j=S_j+h\hat{C}_i$
end
end
 $P_n(x_0)=\sum_{i=1}^p S_j$.

In many practical problems, Lagrange interpolating polynomials may not be suitable for use as an approximation. This is because polynomials of high degree often have a very oscillatory behavior, which is not desirable in approximating functions that are reasonably smooth. One alternative to interpolation is to find a polynomial of a low degree that "best fit" the data, in which, piecewise polynomial interpolation is an attractive one. In piecewise polynomial interpolation, several lower-degree polynomials are joined together in a continuous fashion so that the resulting piecewise polynomial interpolates the data. One of the most commonly used piecewise interpolation methods is the Lagrange piecewise cubic interpolation.

Let y = f(x) be a continuous function in [a, b]. The Lagrange piecewise cubic function interpolates f(x) on the nodes $x_j = a + jh$ for j = 0, 1, ..., n and h = (b - a)/n. Thus, we obtain n - 2 cubic polynomials

$$P_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$$

for i = 1, ..., n-2. Substitute $(x_{i-1}, y_{i-1}), (x_i, y_i), (x_{i+1}, y_{i+1}),$ and (x_{i+2}, y_{i+2}) into the Lagrange polynomial form, and define

$$\Delta_{11} = \frac{1}{6h^3} (x_i + x_{i+1} + x_{i+2}),$$

$$\Delta_{12} = \frac{1}{2h^3} (x_{i-1} + x_{i+1} + x_{i+2}),$$

$$\Delta_{13} = \frac{1}{2h^3} (x_{i-1} + x_i + x_{i+2}),$$

$$\Delta_{14} = \frac{1}{6h^3} (x_{i-1} + x_i + x_{i+1}),$$

$$\Delta_{21} = \frac{1}{6h^3} (x_i x_{i+1} + x_i x_{i+2} + x_{i+1} x_{i+2}),$$

$$\Delta_{22} = \frac{1}{2h^3} (x_{i-1} x_{i+1} + x_{i-1} x_{i+2} + x_{i+1} x_{i+2}),$$

$$\Delta_{23} = \frac{1}{2h^3} (x_{i-1} x_i + x_{i-1} x_{i+2} + x_i x_{i+2}),$$

$$\Delta_{24} = \frac{1}{6h^3} (x_{i-1} x_i + x_{i-1} x_{i+1} + x_i x_{i+1}),$$

$$\Delta_{31} = \frac{1}{6h^3} x_i x_{i+1} x_{i+2},$$

$$\Delta_{32} = \frac{1}{2h^3} x_{i-1} x_{i+1} x_{i+2},$$

$$\Delta_{33} = \frac{1}{2h^3} x_{i+1} x_i x_{i+2},$$

$$\Delta_{34} = -\frac{1}{6h^3} x_{i-1} x_i x_{i+1},$$

where Δ_{ij} for i = 1, 2, 3, 4 and j = 1, 2, 3, then the coefficients of a_i, b_i, c_i and d_i for i = 1, ..., n-2 may be defined in following matrix multiplication form

$$A = BY \tag{21}$$

where

$$A = \begin{pmatrix} a_1 & a_2 & \dots & a_{n-2} \\ b_1 & b_2 & \dots & b_{n-2} \\ c_1 & c_2 & \dots & c_{n-2} \\ d_1 & d_2 & \dots & d_{n-2} \end{pmatrix},$$

$$B = \begin{pmatrix} \frac{1}{6h^3} & \frac{1}{2h^3} & -\frac{1}{2h^3} & \frac{1}{6h^3} \\ \Delta_{11} & \Delta_{12} & \Delta_{13} & \Delta_{14} \\ \Delta_{21} & \Delta_{22} & \Delta_{23} & \Delta_{24} \\ \Delta_{31} & \Delta_{32} & \Delta_{33} & \Delta_{34} \end{pmatrix},$$

and

$$Y = \begin{pmatrix} y_0 & y_1 & \cdots & y_{n-3} \\ y_1 & y_2 & \cdots & y_{n-2} \\ y_2 & y_3 & \cdots & y_{n-1} \\ y_3 & y_4 & \cdots & y_n \end{pmatrix}.$$

The parallel processing for the matrix multiplication (21) is straight forward. Matrix Y is distributed among p processors in a multicomputer, and p set of columns of matrix B are assigned to the each processor. Then the coefficients of a_i , b_i , c_i and d_i for i=1, ..., n-2 are computed in parallel on p processors. The evaluation of each cubic function may also be done on p processors in parallel by Horner's rule

$$y_i(x) = ((a_ix + b_i)x + c_i)x + d_i$$

for i = 1, ..., n - 2.

6 Conclusions and future work

We have discussed parallel methods for polynomial evaluation, polynomial multiplication and polynomial division and Lagrange piecewise cubic interpolation, and presented some of the experimental results on the Intel hypercube. These methods may be easily implemented on a distributed memory multicomputer. We expect the methods will gain good speedups' on a distributed memory multicomputer with careful data distribution. The next step of the work is to implement the rest of the methods on a local memory multicomputer to test the performance. We also plan to investigate other parallel methods for polynomial related problems, such as multi-solutions of a polynomial, different polynomial interpolation methods, cubic spline interpolation and others.

References

Aho, A., Hopcroft, J., and Ullman, J. [1974], The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA.

Bini, D. [1984], "Parallel solution of certain Toeplitz linear systems", SIAM J. Comput., Vol. 13, No. 2, pp. 268-276.

Chamberlain, R. [1986], "Grey codes, Fast Fourier Transform and hypercubes", *Technical Report* 864502-1, Chr. Michelsen Institute. Bergen, Norway.

Chan, T. [1986], "On grey code mappings for mesh-FFTs on binary n-cubes", *Technical Report* 86.17, RIACS, NASA Ames Research Center.

Cooley, J. and Tukey, J. [1965], "An algorithm for the machine computation of complex Fourier series, Math. in Comput., 19, pp. 297-301.

Fateman, R. [1974], "Polynomial multiplication, powers and asymptotic analysis: some comments", SIAM J. Comput., Vol. 3, No. 3, pp. 196-213.

Li, G. and Coleman [1988], "A parallel triangular solver for a distributed-memory multiprocessor", SIAM J. Sci., Stat. Comput., Vol. 9, No. 3, pp. 485-502.

Li, G. and Coleman [1989], "A new method for solving triangular systems on distributed-memory message-passing multiprocessors", SIAM J. Sci. Stat. Comput., Vol. 10, No. 2, pp... 382-396.

Lu, H. [1988], "On parallel algorithms for polynomial problems", manuscript, Xian Jiaotong University, China.

Moler, C. [1987], "Matrix computation on distributed memory multiprocessors", *Technical Report*, Intel Scientific Computers.

Norton, A. and Silberger, A. J. [1982], "Parallelization and performance analysis of the Cooley-Tuke," FFT algorithm for shared-memory architectures", *IEEE Transactions on Computers*, Vol. C-36, No. 5, pp. 581-591.

Ponder, C [1988], "Evaluation of "performance enhancements" in algebraic manipulation systems, Ph. D. Thesis, University of California at Berkeley.

Sai, Y., Li, X. and Xie, T. [1986], Synchronous Parallel Algorithms, Science and Technology University of National Defense Press, Hunan, China.

Siebert-Roch, F. and Muller, J. [1989], "VLSI manipulation of polynomials", Computer Algebra and Parallelism, edited by J. Dora and J. Fitch, Academic Press, pp. 233-256.

Siva, C. and Murthy, R. [1989], "Synchronous and asynchronous algorithms for evaluating polynomials on a tree machine", Proceedings of the Fourth International Conference on Supercomputing, Vol. II, pp. 177-179.

You, Z. [1983], Fast Computing Methods for Linear Algebra and Polynomials, Shanghai Science and Technology Press.

Applications of Adaptive Data Distributions *

Eric F. Van de Velde Applied Mathematics 217-50 Caltech Pasadena, CA 91125 Jens Lorenz
Department of Mathematics and Statistics
The University of New Mexico
Albuquerque, NM 87131

Abstract

Continuation methods compute paths of solutions of nonlinear equations that depend on a parameter. This paper examines some aspects of the multicomputer implementation of such methods. The computation is done on the Symult Series 2010 multicomputer.

One of the main issues in the development of concurrent programs is load balancing, achieved here by using appropriate data distributions. In the continuation process, a large number of linear systems have to be solved. For nearby points along the solution path, the corresponding system matrices are closely related to each other. Therefore, pivots which are good for the LU-decomposition of one matrix are likely to be acceptable for a whole segment of the solution path. This suggests to choose certain data distributions that achieve good load balancing. In addition, if these distributions are used, the resulting code is easily vectorized.

To test this technique, the invariant manifold of a system of two identical nonlinear oscillators is computed as a function of the coupling between them. This invariant manifold is determined by the solution of a system of nonlinear partial differential equations that depends on the coupling parameter. A symmetry in the problem reduces this system to one single equation, which is discretized by finite differences. The solution of this discrete nonlinear system is followed as the coupling parameter is changed.

1 Introduction

Concurrent programming is difficult and needs to be simplified. This simple statement describes a major goal of research into concurrent computing. The focus on simplification is justified, because the accumulated experience of earlier feasibility studies is overwhelmingly positive. These feasibility studies required machine-dependent program and problem reformulation. To raise the concurrent technology from the level of feasible to that of usable, much of current research focuses on simplification of the concurrent-programming task.

At the heart of most efficient concurrent programs is data locality: the data is stored in memory locations "near" the processor using the data. To achieve data locality, a data distribution must be introduced. In general, that is a task the programmer must perform, because the best data distribution is determined by global considerations not accessible to analysis by low-level system components (hardware, operating system, and compiler).

This is illustrated by the following simple example of matrix-vector multiplication. Let A be an $M \times N$ matrix, and \mathbf{x} and \mathbf{y} vectors of dimension N and M, respectively. The assignment:

$$\mathbf{y} := A\mathbf{x} \tag{1}$$

requires the evaluation of a matrix-vector product. If this were a self-contained program, not part of a larger program, the optimal data distribution and corresponding optimal program is easily derived. The rows of the matrix A should be distributed evenly (within divisibility constraints) over all concurrent processes. The resulting program is optimal, because it is perfectly load balanced and it requires no communication. Similarly, for the assignment:

$$\mathbf{z}^T := \mathbf{y}^T A \tag{2}$$

one should distribute the matrix columns. For a composite program that evaluates both assignments (1) and (2) neither distribution is optimal. The best distribution distributes both rows and columns; moreover, the process grid is a function of the ratio of the number of times (1) versus (2) is evaluated. Only the user can have a reasonable estimate of this last quantity; hence, only the user can determine the best distribution. (We have ignored the distribution of the vectors for ease of exposition; the conclusion remains valid if one includes them.)

Supplying the data distribution is thus a user task.

^{*}This research is supported in part by Department of Energy Grant No. DE-AS03-76ER72012. This material is based upon work supported by the NSF under Cooperative Agreement No. CCR-8809615. The government has certain rights in this material.

Considering our goal to simplify concurrent programming, supplying the data distribution should be the only concurrency-related user task. Programming languages that allow postponing decisions about data-distribution are under development, see, e.g., Chen [2]. The concept, however, is independent of particular notations or languages, and it can be evaluated within existing concurrent computing systems (although some overheads are to be expected as a result). The program discussed in the remainder of this paper is a realistic illustration of such an approach to concurrent computing, where the data distribution is imposed only after the program was fully developed. In spite of the restriction that all ideas had to be implemented at the software level, instead of at the language or compiler level, excellent performance was obtained. To obtain the best performance, a dynamic data distribution is introduced, which is periodically adapted to achieve global load balance. In this respect, our approach differs significantly from conventional parallelization strategies that break up programs into small "manageable" pieces, typically program loops, and consider each as an independent entity.

An outline of the paper follows. The mathematical aspects of continuation and its application to the computation of invariant manifolds is discussed in section 2. These aspects are covered only to the extent necessary for understanding the algorithmic aspects of the program. For a more detailed treatment, see [12]. In section 3, we discuss the implementation of the program, and in section 4 its performance.

2 Continuation and Invariant Manifolds

Consider a system of M equations:

$$G(\mathbf{u},\lambda) = \mathbf{0} \tag{3}$$

for $\mathbf{u} \in \mathbb{R}^{M}$, which depends on a parameter $\lambda \in \mathbb{R}$. Here,

$$G: \mathbb{R}^M \times \mathbb{R} \longrightarrow \mathbb{R}^M$$

is a smooth map. By a solution branch we mean a one parameter family

$$(\mathbf{u}(s), \lambda(s)) \in \mathbb{R}^M \times \mathbb{R}, \quad s_a \le s \le s_b$$
 (4)

of solutions of (3) depending smoothly on some parameter $s \in [s_a, s_b]$. Because of the importance for applications, many numerical methods have been devised and investigated to compute such branches [6,8]. Assuming that the branch (4) contains only regular points and folds, one has to solve linear systems whose matrices have the form:

$$\begin{bmatrix} A & \mathbf{b} \\ \mathbf{c}^T & \delta \end{bmatrix} \in \mathbb{R}^{(M+1)\times(M+1)} \tag{5}$$

where A is $M \times M$. The matrix A is singular at folds; the bordered system, however, is well conditioned.

We use two concurrent solution methods for such bordered systems. Our first method is a variant of Keller's bordering algorithm [7] that takes into account the possible singularity of the matrix A. The second method is a variant of Goovaerts [5]. Here, we consider only the first method, which begins by computing an LU-decomposition of A. Because the matrix A may be singular, partial pivoting is often not sufficient, and a more general pivoting strategy must be used. For simplicity, the only dynamic pivoting strategy considered here is complete pivoting, but other dynamic strategies are easily substituted. Once the LU-decomposition of A is known, the bordered system is solved using slightly modified back-solves and the solution of a 2×2 system.

Numerical and performance results are given for a — rather involved — test problem, namely the numerical calculation of the invariant manifold of a parameter dependent dynamical system:

$$\frac{d\mathbf{v}}{dt} = F(\mathbf{v}, \lambda). \tag{6}$$

Here, $\mathbf{v}(t) \in T^2 \times \mathbb{R}^2$ and F is a mapping from $T^2 \times \mathbb{R}^2 \times [0, \lambda_0]$ into \mathbb{R}^4 . (With T^2 we denote the standard 2-torus.) The specific example that we have treated is a system of two nonlinear coupled oscillators, where

$$\mathbf{v} = [\theta_0, \theta, \mathbf{r_0}, \mathbf{r_1}]^T$$

and

$$F = \begin{bmatrix} \omega \\ \omega \\ r_0(1 - r_0^2) \\ r_1(1 - r_1^2) \end{bmatrix}$$

$$+ \lambda \begin{bmatrix} -\cos 2\theta_0 + \frac{r_1}{r_0} \left(\cos(\theta_0 + \theta_1) - \sin(\theta_0 - \theta_1)\right) \\ -\cos 2\theta_1 + \frac{r_0}{r_1} \left(\cos(\theta_0 + \theta_1) - \sin(\theta_1 - \theta_0)\right) \\ r_1 \left(\sin(\theta_0 + \theta_1) + \cos(\theta_0 - \theta_1)\right) - r_0(1 + \sin 2\theta_0) \\ r_0 \left(\sin(\theta_0 + \theta_1) + \cos(\theta_0 - \theta_1)\right) - r_1(1 + \sin 2\theta_1) \end{bmatrix}$$
(The value of ω is -0.55 in our calculations.) See Aronson

(The value of ω is -0.55 in our calculations.) See Aronson, Doedel, and Othmer [1] for a motivation of this system and for the study of many interesting bifurcation phenomena. Also, see Dieci, Lorenz, and Russel [3] for a sequential calculation of some invariant manifolds.

In the uncoupled case, $\lambda = 0$, the system has the attracting invariant 2-torus

$$\mathcal{M}(\lambda=0)=\left\{(\theta,1,1):\theta\in T^2\right\}\subset T^2\times\mathbb{R}^2.$$

It follows from general theory (see Fenichel [4] and Sacker [9]) that the torus persists for a sufficiently small coupling constant λ and that it can be parameterized in the form:

$$\mathcal{M}(\lambda) = \{(\theta, R(\theta, \lambda) : \theta \in T^2\},\$$

where $\theta \longrightarrow R(\theta, \lambda)$ is a function from $T^2 \longrightarrow \mathbb{R}^2$. This vector function $R(\cdot, \lambda)$ is the solution of a first order system of partial differential equations, which depends on λ . These partial differential equations are discretized, and a symmetry is utilized to obtain a finite dimensional system of the form (3).

From general theory one expects that the tori $\mathcal{M}(\lambda)$ loose more and more derivatives as λ increases. The torus "breaks" in a certain λ -region and disappears. The calculations in [3] show breaking at about $\lambda=0.2527$. In [12], we compute a solution branch of the discretized system on a 25×25 grid, and we obtain several fold points of this discrete system between $\lambda=0.2430$ and $\lambda=0.2448$.

3 Implementation

The concurrent efficiency of the bordering algorithm is determined almost exclusively by the efficiency of the LU-decomposition. The latter, in turn, depends crucially on an interplay between the pivot locations and the distribution of the matrix entries over the concurrent processes. In particular, if the pivots are known in advance, the data distribution can be chosen accordingly, and near ideal load balance can be achieved. In this case, the algorithm is also easily vectorized because all active data remain in contiguous blocks.

Hence, efficiency can be obtained with preset pivots, but numerical stability will, in general, require a different pivoting strategy. In our approach these two requirements are hardly in conflict, because many highly correlated matrices are factored in the course of the continuation procedure. The reasonable belief that the pivot locations can be kept constant along a whole piece of the branch is indeed confirmed by our experience. Therefore, both numerical stability and load balance can be achieved by using a dynamic pivoting strategy occasionally (when the growth factor has exceeded some limit), followed by an adaptation of the data distribution to the new pivot locations.

This data distribution strategy differs from most others in two essential aspects. First, it takes into consideration the global behavior of the program, i.e., the fact that the matrices result from a continuation procedure. Second, adapting the data distribution to the computation itself is an integral part of the strategy. In section 4, we shall see that the combination of these two ingredients leads to high efficiency. Here, we consider the implementation aspects of this strategy.

Because the data distribution is adaptive and depends on the global nature of the continuation program, component routines like the LU-decomposition and the backsolve should be written so that they are correct independently of the data distribution. For such routines, the data distribution is part of the input data supplied in the

argument list when calling the routine. We use the LUdecomposition described in [11] and its companion backsolve algorithm. To achieve independence of the data distribution, the LU-decomposition must do all pivoting implicitly (otherwise the data distribution would depend on the pivots!). In fact, all routines called by our program must have the property that they are correct independently of the data distribution. If we consider these routines as the components of a library, the necessity for this property follows from the observation that the writer of the library routines cannot know the global properties of the program in which this routine will be used. Hence, the data distribution cannot be fixed at the time of writing the library. In fact, our LU-decomposition, matrix-vector operations, and other related linear algebra routines are packaged in a data-distribution-independent library. Our continuation program uses this library and imposes a data distribution on it at run-time.

To provide maximum flexibility, our LU-decomposition allows pivoting of both rows and columns. Besides allowing classical pivoting strategies (row, column, diagonal, and complete), this flexibility also leads to two intrinsically concurrent pivoting techniques with increased numerical stability and load balance. For details on those techniques, we refer to [11]. For the discussion of our continuation program we introduce just one dynamic strategy, complete pivoting, and one static strategy, preset pivoting. Complete pivoting is, in general, overkill since numerical stability can be obtained with less expensive pivoting strategies. However, complete pivoting ensures that the pivot locations are highly unpredictable and, hence, illustrates best the adaptivity of our program. Moreover, complete pivoting is used only occasionally, i.e., when the growth factor exceeds a set tolerance. For most LUdecompositions, we use preset pivots, determined by the last LU-decomposition with dynamic pivoting. Hence, the cost of dynamic pivoting is amortized over many LUdecompositions.

4 Performance

The calculations were performed on a Symult Series 2010 multicomputer with up to 64 nodes. We investigate the dependence of the execution time on the data distribution for one LU-decomposition. Here, we used 64 nodes and an 8×8 process grid. As expected, the adapted data distribution turned out to be superior. We consider also, for each fixed strategy, the dependence of the execution time on the number of nodes. We used 2, 4, 8, 16, 32, and 64 nodes, and obtained excellent speedup for each strategy. For absolute performance, we made a comparison of the sequential version of our code with a fully optimized C-version of the LINPACK benchmark [10]. Due

Pivoting	Distrib.	Time(s)	Spdp.	Eff.(%)
Complete	Linear	75.3	41.4	64.7
Complete	Random	63.7	49.9	78.0
Complete	Scatter	62.8	46.2	72.2
Complete	Adapted	51.3	54.2	84.7
Preset	Linear	48.9	36.9	57.7
Preset	Scatter	40.3	42.6	66.6
Preset	Adapted	33.8	48.9	76.4
F. Preset	Adapted	29.7	50.0	78.2

Table 1: LU-Decomposition times for a 25×25 grid problem on 64 node Symult Series 2010. Number of megaFLOPS is based on $M^3/3$ floating point operations, where $M = 25^2$ is the number of unknowns.

to memory restrictions, this comparison was done with a random 300×300 matrix. A sequential version of our fast preset pivoting algorithm ran about 5% slower than LIN-PACK. (These 5% result from the fact that we have not implemented a number of low level optimizations used by LINPACK.)

We consider the example of section 2 with $h = 2\pi/25$, i.e., the number of unknowns at every step is M = 625. In Table 1, we present timings for one (typical) LUdecomposition using complete pivoting and preset pivoting in combination with different data distributions for the factored matrix. The linear and scatter distributions are static distributions. The linear distribution allocates blocks of contiguous rows and columns to processes. The scatter distribution uses a wrap mapping. The adapted distribution uses the pivot locations of the previous LUdecomposition to distribute the current matrix such that ideal load balance is achieved, if the pivot locations of the current matrix coincide with those of the previous matrix. In the version "Fast Preset" of preset pivoting, certain administrative overhead is eliminated using the information that the pivots are preset and that a particular distribution is used. All calculations were done on a 64 node machine using 64 processes, one process running on each node. The process grid was partitioned into P = 8 process rows and Q = 8 process columns.

To test the concurrent performance of our code, we determine the execution time as a function of the number of nodes. The same example as in Study 1 is computed successively using 2, 4, 8, 16, 32, and 64 nodes, and always choosing the number of processes equal to the number of nodes, one process running on each node. The numbers P and Q of process rows and columns were chosen equal within divisibility constraints. When the logarithm of the execution time is plotted as a function of the logarithm of the number of processes, ideal speedup is characterized by a straight line with slope -1 if appropriate scales are used. Figure 1 shows that, for each strategy,

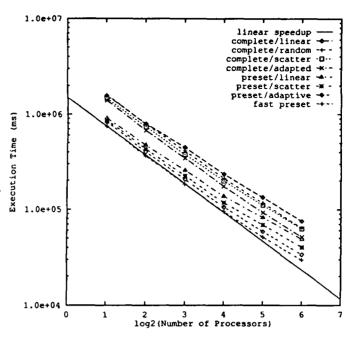


Figure 1: LU-Decomposition times for a 25×25 grid problem as a function of number of nodes on a Symult Series 2010.

the execution-time plot is almost parallel to the line characterizing ideal speedup. Table 1 can be used to identify the individual timing plots.

The problem was too big to run on a one-node machine. Precise speedups could thus not be calculated. In Table 1, we give speedups and efficiencies with respect to two-node timings, i.e., the real speedup is estimated by:

$$S_{PQ} \approx 2 * T_2/T_{PQ}$$

and the real efficiency is estimated by:

$$\epsilon_{PQ} \approx 2 * T_2/(PQT_{PQ}).$$

Here, T_2 is the two-node timing and T_{PQ} is the timing with $P \times Q$ nodes. Speed-up and efficiency are good measures for the overhead due to communication and load imbalance.

When varying the data distribution and keeping the pivoting strategy fixed, it is clear that the adapted data distribution is the most efficient. This is easily explained by the increased load balance of the adapted data distribution. This observation holds for both complete pivoting and preset pivoting.

When comparing efficiencies for the same distribution but for different pivoting strategies (i.e., in Table 1 compare lines 1 and 5, 3 and 6, 4 and 7), it is seen that complete pivoting is more efficient. This is because the pivot-search cost leads to a higher ratio of computation to communication time for complete pivoting than for preset pivoting.

Another interesting observation, which follows from Table 1, is that complete pivoting with the random distribution (line 2) is more efficient than complete pivoting with the scatter distribution (line 3). The execution time, however, is lower for the scatter distribution. The random distribution is better than the scatter distribution for load balancing, and hence, has higher efficiency. The random distribution leads to very irregular memory access patterns, however, and that causes the absolute execution time to be larger.

References

- D. G. Aronson, E. J. Doedel, and H. G. Othmer. An analytical and numerical study of the bifurcations in a system of linearly-coupled oscillators. *Physica*, 25D:20-104, 1987.
- [2] M. Chen, Y.-I. Choo, and J. Li. Compiling parallel programs by optimizing performance. The Journal of Supercomputing, 2:171-207, 1988.
- [3] L. Dieci, J. Lorenz, and R. D. Russel. Numerical calculation of invariant tori. 1989. To appear in SIAM Journal on Scientific and Statistical Computing.
- [4] N. Fenichel. Persistence and smoothness of invariant manifolds for flows. *Indiana University Mathematics Journal*, 21:193-226, 1971.
- [5] W. Goovaerts. Stable Solvers and Block Elimination for Bordered Singular Systems. Report, Rijksuniversiteit Gent, Ghent, Belgium, 1989.
- [6] H.B. Keller. Numerical Methods in Bifurcation Problems. Tata Institute of Fundamental Research, Bombay, 1987.
- [7] H.B. Keller. Practical procedures in path following near limit points. In R. Glowinski and J.L. Lions, editors, Computing Methods in Applied Sciences and Engineering, North-Holland, 1982.
- [8] W.C. Rheinboldt. Numerical Analysis of Parametrized Nonlinear Equations. Wiley, New York, NY, 1986.
- [9] R. Sacker. A perturbation theorem for invariant manifolds and Hölder continuity. *Journal Mathematical Mechanics*, 18:705-762, 1969.
- [10] B. Toy. Private Communication.

- [11] E. F. Van de Velde. Experiments with Multicomputer LU-Decomposition. Report CRPC-89-1, Center for Research in Parallel Computing, 1989. To appear in Concurrency: Practice and Experience.
- [12] E. F. Van de Velde and J. L. Lorenz. Adaptive Data Distributions for Concurrent Continuation. Report CRPC-89-4, Center for Research in Parallel Computing, 1989.

The Quadratic Sieve Factoring Algorithm on Distributed Memory Multiprocessors

Michel COSNARD - Jean-Laurent PHILIPPE
Laboratoire de l'Informatique du Parallélisme - IMAG
Ecole Normale Supérieure de Lyon
46, allée d'Italie 69364 LYON Cédex 07 FRANCE
e-mail: cosnard@frensl61.bitnet

Abstract

The quadratic sieve algorithm is a powerful method for factoring large integers up to 100 digits. In this paper, we study in detail each step of the algorithm, in order to derive efficient parallel implementations on distributed memory multiprocessors. Our aim is to prove, taking the quadratic sieve algorithm as a revealer, that very efficient programming methodologies could be derived in order to take the most out of the target architecture. We describe an implementation of the quadratic sieve algorithm on the FPS T40 hypercube. We evaluate the solution through the results we obtain, and particularly the superlinear speedup. We try to explain these speedups. From these experimental considerations, we derive more efficient implementations, including totally equidistributed tasks among the processors. Some other refinements may be added to the algorithm.

I - Introduction

The quadratic sieve algorithm [Pom 85], [Sil 87] is a powerful method for factoring large integers up to 100 digits. Various authors have already published

This work has been partially supported by the Coordinated Research Program C3 of the French CNRS and the DRET.

interesting results on parallel implementations of this algorithm [PST 88], [CaS 88]. From an algorithmic point of view, these parallel implementations are trivial: the only part of the algorithm to be parallelized is the sieve itself. Indeed, it is the most consuming step, both in time and space, so that most of the parallel versions of the quadratic sieve algorithm have been implemented on networks of independent machines [CaS 88], [Sil 88], [LeM 89]. But some other versions have been implemented on very powerful vector computers, taking advantage of the large memory space and the high power of the processors of the Cray XMP for example [DHS 85], [RLW 88]. Only one attempt is known to us to implement this algorithm on a distributed memory multiprocessor, by Davis and Holdridge [DaH 88], on the Ncube.

We propose to modelize all the steps of the algorithm for an implementation on distributed memory multiprocessors: the initialization phase, the generation of the polynomials, the sieve, the choice of the w(x) candidates to factorization, the factorization of these w(x), and the Gaussian elimination on the w(x) matrix.

In this paper, we study in detail each step of the algorithm, in order to derive efficient parallel implementations on distributed memory multiprocessors. The target architecture is the FPS T40 hypercube, crudely parameterized in terms of communication facilities. The FPS T40 is a 32 processors (T414) hypercube with Weitek floating point coprocessors; the communication is only possible between neighbors of a 5-dimensional

hypercube topology; communication implies local synchronization since it is done through a "rendezvous" protocol.

Our aim is to prove, taking the quadratic sieve algorithm as a revealer, that very efficient programming methodologies could be derived in order to take the most out of the architecture. In the case of the FPS T40, load balancing the computation phases, overlapping computation and communication and restricting the message exchanges to neighbors are the crucial features. Interconnection topologies of communicating processes are reduced to subtopologies of the hypercube.

II - The quadratic sieve algorithm

Suppose that an integer N is not a prime number (this can be verified through a Fermat's test), and N has no small prime factors (they have been found with a sieve of Eratosthenes).

Here is the algorithm to factor N, with the quadratic sieve method:

Begin algorithm

1 - Initialization step:

Compute and store:

k (number of elements in the base),

B (the smallest power of 2 greater than each element of the base),

M (the upper bound of the sieve interval).

Compute and store the k elements pi of the

base, with p_i prime and $\left(\frac{N}{p_i}\right) = 1$.

Compute and store the solutions of the equations $x^2 \equiv N \mod p_i^{\alpha}$, with $p_i^{\alpha} \leq B$.

End initialization phase

While (not enough factored w(x)) do

2 - Compute a polynomial $w(x)=a^2x^2+2bx+c$: Compute a: a must be prime, a = 4k+3,

a near
$$\sqrt{\frac{\sqrt{2N}}{M}}$$
, and $\left(\frac{N}{a}\right) = 1$.

a near $\sqrt{\frac{\sqrt{2N}}{M}}$, and $\left(\frac{N}{a}\right) = 1$. Deduce b: $b = N^{\frac{a(b+1)}{2}+1} \mod a^2$, and $|b| < \frac{a^2}{2}$.

Deduce c:
$$c = \frac{(b^2 - N)}{a^2}$$
.

<u>3 - Sieve</u> with each element p_i of the base and each power α , such that $p_i^{\alpha} \leq B$: Initialize the array tab_sieve[-M, M[to 0. For each p_i^{α} do Find the starting index s near -M.

Repeat Add $\log_2 p_i$ to tab_sieve[x]. $s = s + p_i^{\alpha}$. until $s \ge M$

endfor

4 - Define a bound V.

For each x in [-M, M[with tab_sieve[x] $\geq V do$

Compute w(x).

5 - Try to factor w(x) over the base. Eachfactored w(x) is stored as a line of the matrix M. End for

End while

<u>6 - Perform a gaussian elimination on the</u> matrix M of the w(x).

Compute the GCD of some dependent lines of the matrix M. The GCD is a cofactor of N.

End algorithm.

Figure 1. The MPQS Algorithm.

1 - The initialization step

The initialization step mainly computes the k elements of the base of prime numbers to be used in remaining steps. This computation can be achieved in a full parallel efficiency when using the P processors, indexed by i (i=0...P-1), as a farm of processors. The k elements are statistically lower than 2k log(2k). Each processor i is assigned an

interval
$$\left[i\frac{2k \log(2k)}{P} + 1, (i+1)\frac{2k \log(2k)}{P}\right]$$
. In this

interval, there are statistically $\frac{k}{P}$ elements. The processors search for all the elements in their proper interval.

Once they have completed this step, we gather and sum on processor 0 (in log(P) communication steps) the effective number m of elements that each processor has found. Then, processor 1 searches for the (k-m) missing elements to get a distributed base of k elements, while the other processors equally distribute the k elements of the base on the P-1 remaining processors.

Balancing the elements between the (P-1) remaining processors can be achieved in log P communication steps using a all to one-type gathering algorithm on a spanning tree. Let us describe in more detail the algorithm for a hypercube topology. The following figure shows the second step of this strategy:

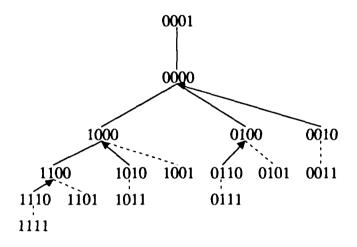


Figure 2. Second step on a a 4-dimensional hypercube

Consider a spanning tree of the hypercube rooted in processor 0, such that processor 1 is the subtree reduced to one element. Isolate processor 1 during its computation phase (computation of the (k-m) missing elements in the base). Each leaf of the tree exchanges some elements with its father in order to get the exact number. Recursively suppress the leaves and repeat the process on a spanning tree of a hypercube of reduced dimension.

After log P communicating steps, all the processors, except processors 0 and 1, have the exact number of elements. As soon as processor 1 has computed the (k-m) missing elements, 0 and 1 load balance their elements. This algorithm works for any initial distribution of the number of elements we have encountered, thanks to the distribution of primes in intervals.

Let us present the TNode and the iPSC distributed memory multiprocessors. The TNode computer has 32 to 128 processors (T800 Transputer) with on chip floating point arithmetics; a dedicated network allows dynamic reconfiguration of the interconnection topology (network of maximum degree 4); reconfiguration implies global synchronization points; communication implies local synchronization since it is done through a "rendez-vous" protocol.

The iPSC has 32 to 64 processors (80286) with floating point coprocessors; routing VLSI devices allow for point to point communications between the processors of the hypercube; messages are stored in a FIFO queue for each processor, which implies no synchronization although the exchanges must be done carefully.

This technique is well suited for the FPS T40 computer since it is basically a hypercube. In the case of the TNode, the network has to be configurated in such a topology that its diameter be of order of log P. For example, we could use a perfect shuffle ring network. In the case of the iPSC hypercube, this technique works when taking care of the amount of data transmitted.

The next step consists in computing the roots of the equations $x^2 \equiv N \mod p_i^{\alpha}$, for each element p_i of the base, and each integer α such that $p_i^{\alpha} \leq B$. Before computing these roots, we have to distribute the p_i 's such that the computation of the roots takes the same time on all the processors. Typically, each p_i implies α equations to be computed. For large p_i 's, $\alpha = 1$. But for small p_i 's, $\alpha > 1$. So, if each processor has

the same number $\frac{k}{P}$ of elements, the processors with the smallest values of the p_i 's have more work to perform than the other processors.

This is the reason why we slightly modify the step for the repartition of the k elements of the factor base. One still computes the total number of equations to be solved. And instead of balancing the k elements over the P processors, one distributes the elements such that the number of equations is the same on each processor. But to solve $x^2 \equiv N \mod$ p_i^{α} , with $\alpha \ge 2$, one needs the solution of $x^2 = N$ $\text{mod } p_i^{\alpha-1}$. So all the equations for a given p_i must be solved on the same processor. Otherwise some communications occur and waste time. And it is not efficient anymore. Hence, under the constraint that the equations for a given p; are solved on a single processor, the tasks may not be strictly balanced. But each processor can perform independently and with maximum efficiency this computation. The roots are stored on each processor.

The computation within the initialization phase is all done. Depending on the requirements of the sieving phase of the algorithm, one has to broadcast the parts

of the base from each processor to all the other ones. As the amount of data to be broadcast is large, one can use the broadcast algorithms in an hypercube given by Ho and Johnsson, to achieve low communication costs [HoJ 89].

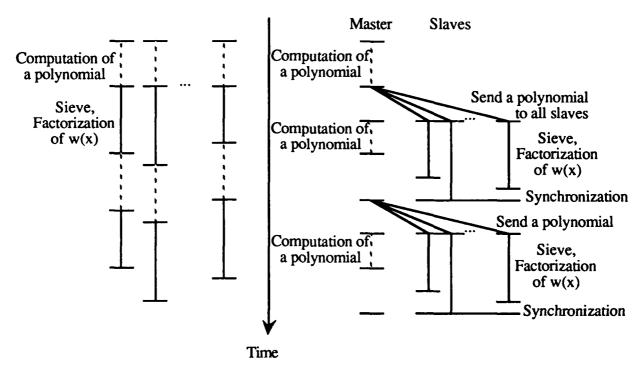
2 - The generation of the polynomials

The polynomials in the quadratic sieve are used to generate integers that factor on the base. There are many ways for generating the polynomials on a distributed multiprocessor, depending on the way both the elements of the base and the sieve interval are distributed among the processors.

- The polynomials can be generated before the loop. In this case, the generation can be done with full parallelism, such that no communication occurs, and no processor is idle while other ones are working. The polynomials are present in the network but they are distributed among the processors. They take a large space (680 Mbytes for a 100-digit N). This is unrealistic for our examples of distributed memory multiprocessors.
- Or they can be generated in the sieving loop.
 - The polynomials can be generated by each processor. But redundant work is done, if the processors have parts of the base and/or parts of

- the sieve interval, because all of them have to generate the same polynomials.
- The polynomials can be generated by a dedicated master processor. This master sends the polynomials to the slaves that sieve. We have to decide whether the slaves have the whole base and the whole sieve interval, or if one of these entities or both are distributed among the slaves. If they have the whole base and the whole interval, each slave must sieve with its proper polynomials. So the master must send different polynomials to each slave (many sequential communications). In the other cases, the master must send the same polynomial to each slave (a single one_to_all communication).

It is not clear to decide which solution needs the minimum execution time (see figure 3 below). In the farm of independent processors (1), a compensation factor may appear. Indeed, the time needed for computing a new polynomial is the same on all the processors. But the time needed for sieving may be slightly different. And this cannot occur in the Master/Slaves solution (2): of course there is no redundant work, but we can see that communications and synchronizations will never let compensation



(1) Independent processors

(2) Master / Slaves

Figure 3. Different timings for the execution with or without a Master

play its role... Indeed, with the Master / Slaves strategy, the time needed to do all the work with one polynomial is the time needed by the slowest processor to do its computation. And furthermore, only P-1 processors sieve if a processor is dedicated to the computation of the polynomials, but P processors sieve with the first strategy, because none of them is dedicated to another task.

3 - The sieve

This step is high time consuming. Its implementation depends on the distribution of the elements of the base, the sieve interval, and the polynomials. For a 100-digit integer, the base takes 1 Mbytes and the sieve interval 13 Mbytes. These values are theoretical and depend on the target machine. The polynomials are generated one at a time.

The implementation of [CaS 88] is such that each processor is a workstation and can handle the whole base and the interval. The idea is to give each workstation different polynomials on distributed memory multiprocessors. We cannot really work this way, because of memory requirements. But we can communicate between processors and/or store and recall data on disks. For example, the FPS T40 distributed memory hypercube multiprocessor has 32 1-Mbyte nodes. It is impossible to place both the base and the interval on each processor. We have to distribute them and/or store them on external disks. When a processor needs data, it can either initiate a communication with a neighbor, or read from a disk if the data is too far in terms of communications. The choice depends also on the communication speed between processors and the speed between a processor and a disk.

On the FPS T40, the system and the program use 250 Kbytes on each node. We have 750 Kbytes to be distributed between a sub-interval and a subset of the elements of the base. The best theoretical compromise is to have a large sub-interval and a reduced subset of the elements of the base. This implies the minimum total communication time. Whatever the base and the interval divide the 750 Kbytes, communicating between neighbors is always faster than reading data from disks.

4 - The choice of the w(x) candidates for a complete factorization

This step is a linear search in the interval sieve to find those w(x) that will probably factor on the base. If the interval is completely distributed among the processors or if each processor has the whole

interval (sieved with different polynomials on different processors), this step can be achieved in a total parallel way. It is completed for each new polynomial.

5 - The factorization of these w(x)

If each processor has the whole base, this step can be achieved in a parallel manner, without any interaction between processors. If not, a pipeline algorithm can be implemented, such that each w(x) be divided by all the elements of the base (the elements are distributed among the processors). This step is done using a second sieve. A possible implementation is to use a logical ring and to synchronously rotate the information from one processor to the other. The smallest amount of data has to be communicated: a choice is necessary between the sub-interval and the subset of the elements of the base.

6 - Gaussian elimination

The solution of large dense linear systems of algebraic equations in finite fields appears to be the computational kernel of many important algorithms of computer algebra. In particular, the large integer factoring routines in their final stage, such as the quadratic sieve algorithm, compute the nullspace of the transpose of a matrix A, i.e. the vectors x solution of the equation $^{t}x.A = 0$. The quadratic sieve algorithm requires the factorization of a matrix of more than 30,000 rows and columns for a 100-digit integer. However, gaussian elimination requires an amount of arithmetic operations proportional to the cube of the order of the matrix, and this leads to serious limitations both on the size of the problems that can be dealt with, and the speed of the solution, when implemented on sequential computers. An implementation is possible on a linear array of processors [CoR 87].

III - Implementation on the FPS T40 hypercube

The FPS T40 hypercube is viewed as a network of transputers, since we do not use the facilities provided by the Weitek coprocessors. The programs are written with the C language, allowing for dynamic use of the memory. All the processors have almost the same program code but work on different data. The infinite integer precision package is due to J.L. Roch [Roc 90]. It is very powerful and we use it during the initialization phase, during the sieve phase for generating new polynomials and factoring

some w(x) and for computing gcd's at the end of the execution.

We do not implement any of the algorithmic or mathematical variations that can be found in the literature.

1 - Implementation

Concerning the initialization phase, we implement the theoretical study presented above. Each processor receives a subinterval of the total interval where the k elements of the base may statistically be. Each processor searches for all the elements in its set. Then, a reduction phase occurs in order to count the number of found elements. On a spanning tree embedded in the hypercube, all the processors send to their father their elements number. The father sums the numbers and sends them to its father, up to the root.

The root designates a processor PE (with no sons in the tree) which searches for all the missing elements. Then each processor sends to or receives from its father some elements in order to get the right number of elements. This is necessary to equidistribute the work for the computation of the square roots of N. When processor PE has computed all the missing elements, it exchanges some of them with the root and thus both of them get the right number of elements. That is why computing the square roots of N can be then executed with maximum efficiency, because it takes the same time for all the processors.

We can say now that the factor base (elements and square roots) is completely computed. But it is distributed among the processors. Our implementation requires that each processor knows the whole base. This means that each processor has to send its part of the base to all the other processors, and has to receive from all the other processors their own part of the base. This can be done through a all_to_all communication procedure. Then the whole factor base is present on each processor. This implies a strong redundancy, but avoids future communication steps.

The sieve phase is implemented as follows. We distribute the polynomials among the processors. Hence each processor has a proper family of polynomials, the whole factor base (which is already the case), and the whole sieve interval. This interval is viewed as an array indexed on [-M, M[. During this phase, the processors can work independently as a farm of processors. Each of them has to find k/32

lines of the matrix, in order to get k lines among all the processors. In fact, about 0.96k lines may suffice. The processors do not need to communicate since they all perform a complete quadratic sieve factorization, but with a restricted set of polynomials. Of course each one of them generates a small part of the whole matrix. All the parts are gathered at the end of the execution to build the global matrix M of the prime decomposition of the factored w(x).

This matrix is embedded in $\mathbb{Z}/2\mathbb{Z}$. A gaussian elimination is performed. And with a probability 0.5, each linear dependency leads to a non trivial cofactor of N.

2 - Results

We have factored small integers in the range 35-41 decimal digits. The execution times are given on the following figure 4.

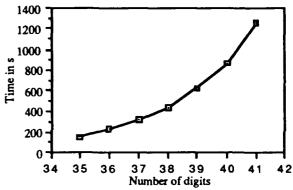


Figure 4. Execution time vs size of N

Figure 4 shows that this algorithm is not linear in regard of the number of digits of N. In fact, the heuristic run time is [Pom 82]:

$$\exp((1+o(1))\sqrt{\ln N \ln \ln N})$$

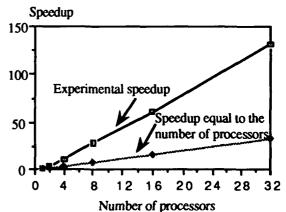


Figure 5. Speedups for a 41-digit integer

In order to compute experimental speedups, we tried to factor a 41-digit integer. Figure 6 depicts the experimental speedup curve compared with the linear speedup. The obtained performance is surprising: with 32 processors, the speedup (computation time with one processor compared with computation time with 32 processors) is equal to 132.

Number of processors	Speedup
1	1
2	3.4
4	11.2
8	28.6
16	60.4
32	131.8

Figure 6. Speedup when factoring a 41-digit integer

The superlinear speedups come from the growth of memory space when increasing the number of processors. As the dynamic memory space is mainly used to compute the polynomials, increasing the memory space avoids parts of the management of this memory, i.e. less "allocate" and "free" procedure calls.

3 - Load balancing the computation

We can see on more acurate results that the work is not equally distributed among the processors. Indeed, the slowest processor needs twice the time of the fastest processor to complete its task. Furthermore, we can see that this time is directly linked to the number of polynomials that this processor has to generate and work with, to get its k/32 lines. The following figures 7 and 8 show that the ratio of the slowest processor time to the fastest processor time is close to 2. The workload is 75 %.

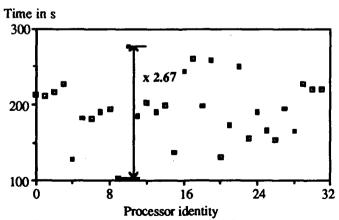


Figure 7. Time to compute polynomials per processor

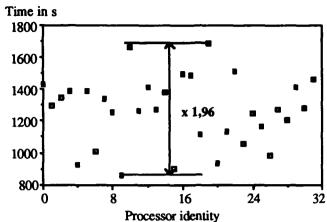


Figure 8. Total execution time per processor

In order to improve the workload, we have to load balance the computation by letting the processors work with the same number of polynomials.

If we can know a priori how many polynomials, say G, are required to get the k lines, each processor may work with G/32 polynomials. Caron and Silverman [CaS 88] have experimentally studied the size of the factor base, the size of the sieve interval and the number of polynomials necessary for the factorization of a n-digit integer. Using these informations, we can compute an approximate value of G (Figure 9).

Number of polynomials

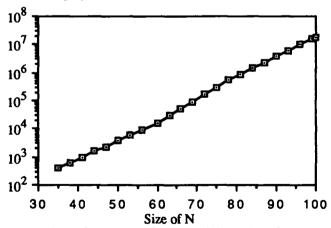


Figure 9. Number of polynomials vs size of N

Hence the improved algorithm computes first the number of polynomials necessary to the factorization and then each processor works with G/32 polynomials in order to get the desired total number of rows.

Using this new strategy, the difference between the slowest and the fastest processors is drastically reduced: each step of the algorithm is well balanced on each processor.

Figure 10 shows the factorization time for larger values of N in the range 35-60 decimal digits.

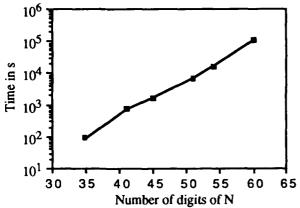


Figure 10. Execution time in s vs size of N

(For N with about 60 digits, it takes approximately 31.5 hours to factor).

The percentage of the time during which the fastest processor is idle while at least one processor is still working is shown on the following figure 11.

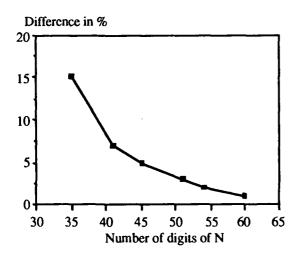


Figure 11. Difference of time (%) between the slowest and the fastest processors

The workload is better. The following table gives the execution time of the slowest and fastest processors and the workload for integers in the range 41-60 digits.

Size of N	Slowest processor	Fastest processor	Workload
41	780 s	836 s	96.5 %
45	1,620 s	1,700 s	97.4 %
51	6,552 s	6,742 s	98.6 %
54	18,152 s	18,548 s	99.1 %
60	112,284 s	113,613 s	99.5 %

Figure 12. Execution times and workload

IV - Comparison FPS T40, iPSC, TNode

Each node of the iPSC/2 hypercube [Arl 88] is a 80286 processor with a 80287 coprocessor. It can have 4 Mbytes memory per node. Suppose we have an iPSC with 32 nodes.

To compare an implementation on the iPSC/2 and on the FPS T40, one must say that the Transputer T414 and T800 are RISC microprocessors (7.5 Mips) and the 80286 is a CISC microprocessor (4 Mips). For our application, one can say that the 80286 is more powerful by a factor approximately 2. But the most important feature is the memory space. As the generation of the polynomials is high dynamic memory consuming, we can say that multiplying the memory space by a factor 4, will allow to speed up the execution by a factor of 6. Both these speedups together will bring a speedup of about 12 on the iPSC, compared to the FPS T40. This means that we can factor integers of the same size, 12 times faster on the iPSC/2 than on the FPS T40, or we can factor a 70-digit integer on the iPSC/2 within the time needed for a 60-digit integer on the FPS T40.

The TNode multiprocessor [ChT 89] is not an hypercube. But with the dynamic reconfiguration facilities, it is possible to use it as a tree, for the initialization phase, for example. The TNode is based on the T800 Transputer, which has approximately the same features as the T414 Transputer, for what concerns integer arithmetic. It is more powerful when using floating-point arithmetic. But we do not need these facilities. So a TNode with 32 nodes and 1 Mbytes memory per node will not be much faster than the FPS T40. But by using a MegaNode (128 nodes), it is clear that we can reach higher limits, because of the memory increase.

V - Conclusion

Siverman [Sil 88] noted that in certain cases, one can obtain better than linear speedup by partitioning an algorithm among machines due to effects such as much greater real memory and having multiple data caches. Indeed even on a tightly coupled MIMD massively parallel architecture, such speedups can be reached. This clearly shows the importance of the memory requirements for the MPQS algorithm. However, with 1 Mbytes of memory per processor, efficient results can be obtained.

Our results compare favorably with those obtained by Davis and Holdridge [DaH 88] on the NCube of 1,024 nodes, each with 512 Kbytes of local memory: the difference of time is bounded by a factor of 2.

We are implementing a completely parallelized version of the quadratic sieve algorithm on the FPS T40 target architecture. First experiments show that this implementation could be done in very efficient ways. In particular, it is worth noting that the bulk of computation work resides in the sieve itself (§ 3). This part must be carefully handled and the most efficiently parallelized. For a 100-digit integer it seems to represent about 70% of the work. If the polynomials are generated by the processors in the sieve loop, a farm of processors could be used with high efficiency and very low redundant work, but some redundant data.

But some other refinements are possible (algorithmic and mathematical refinements). They are currently under implementation, and will drastically decrease the total execution time.

References

[Arl 88] Arlauskas S., "iPSC/2 System: A Second Generation Hypercube", Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications, ed. G.C. Fox, ACM, 1988, pp. 38-42.

[CaS 88] Caron T. R., Silverman P., "Parallel Implementation of the Quadratic Sieve", *The J. of Supercomputing*, 1, 1988, pp. 273-290.

[ChT 89] Champion T., Tourancheau B., "TNode: document utilisateur", Technical Report 89-02, LIP, ENS Lyon, 1989.

[CoR 87] Cosnard M., Robert Y., "Implementing the Nullspace Algorithm over GF(p) on a Ring of Processors", Second Int. Symp. on Computer and Information Sciences, E. Gelenbe & A. Riza Kaylan eds., Bogazici University, Istanbul, 1987, pp. 92-110.

[DaH 88] Davis J. A., Holdridge D. B., "Factorization of Large Integers on a Massively Parallel Computer", Eurocrypt '88 Abstracts, A Workshop on the Theory and Application of Cryptographic Techniques, IACR, 1988.

[DHS 85] Davis J. A., Holdridge D. B., G. J. Simmons, "Status Report on Factoring", in *Advances in Cryptology*, Lecture Notes in Computer Science, 209, 1985, pp. 183-215.

[HoJ 89] Ho C. T., Johnsson S. L., "Distributed Routing Algorithms for Broadcasting and Personalized Communication in Hypercubes", IEEE Trans. on Computers, Vol 38, n° 9, 1989.

[LeM 89] Lenstra A. K., Manasse M. S., "Factoring by Electronic Mail", *Proceedings Eurocrypt '89*.

[Pom 82] Pomerance C., "Analysis and Comparison of Some Integer Factoring Algorithms", in "Computational Methods in Number Theory", H.W. Lenstra and R. Tijdeman (eds), Mathematisch Centrum, Amsterdam, 1982, pp 89-140.

[Pom 85] Pomerance C., "The Quadratic Sieve Factoring Algorithm", Advances in Cryptology (T. Beth, N. Cot and I. Ingemarrson, eds), Lecture Notes in Comput. Sc., vol 209, Springer Verlag, 1985, pp. 169-182.

[PST 88] Pomerance C., Smith J. W., Tuler R., "A Pipeline Architecture for Factoring Large Integers with the Quadratic Sieve Factoring Algorithm", SIAM J. Comp., vol 17, n° 2, April 1988, pp. 387-403.

[RLW 88] te Riele H. J., Lioen W. M., Winter D., "Factoring with the Quadratic Sieve on Large Vector Computers", Report NM-R8805, Centrum voor Wiskunde en Informatica, Amsterdam, July 1988.

[Roc 89] Roch J. L., "Calcul Formel et Parallélisme. L'Architecture du Système PAC et son Arithmétique Rationnelle", PhD Thesis, Grenoble, december 1989.

[Sil 87] Silverman R. D., "The Multiple Polynomial Quadratic Sieve", *Math. of Comp.*, vol 48, n° 177, January 1987, pp. 329-339.

[Sil 88] Silverman R. D., "Factoring Large Integers in Parallel", ICS, vol 2, 1988, pp. 488-497.

Parallel Quasi-Newton Methods for Unconstrained Optimization

Charles Herbert Still

Parallel Supercomputer Initiative University of South Carolina Columbia, SC 29208

Abstract

This paper describes work done on the 1024 node NCUBE hypercube at the University of South Carolina in developing methods for efficient local solution of unconstrained minimization problems. The paper begins with a mathematical discussion of quasi-Newton methods for unconstrained optimization, and specifically Broyden's Method. Next it presents the parallel methods, and discusses the parallel implementation of the most common Broyden method. Finally it lists some numerical results to evaluate the performance of the parallel Broyden methods.

Introduction

Many types of problems have benefited from the use of high speed parallel processors. The inherent parallelism in these problems has been exploited in order to achieve solutions with the same order of accuracy (or better), shorter times to solution, and/or solutions to problems that would have been intractable on a conventional computer. Examples of these types of problems include fluid dynamics, particle mechanics, and linear systems. One type of problem that has not been as well studied is nonlinear optimization (or its companion problem nonlinear systems of equations).

Traditionally, quasi-Newton methods have been employed to find approximate solutions by iteration. These methods yield high order accuracy, and provide superlinear convergence in a neighborhood of the solution. The most promising quasi-Newton methods are dubbed "secant" methods since they follow a secant line through the previous iterate to select the next iterate.

Let $f: \Omega \subset \mathbb{R}^n \to \mathbb{R}$ be a convex C^2 function. Assume that there is some $x_* \in \Omega$ for which $f(x_*) \leq f(x)$ for all $x \in \Omega$. The usual implementation of a secant method (with rank-one update) to find the minimizer x_* of the smooth function f is

$$H_c s_c = -g_c \tag{1}$$

$$x_+ = x_c + s_c \tag{2}$$

$$H_+ = H_c + u v^T \tag{3}$$

where x_c is the current iterate, g_c is the gradient of f at the current iterate, H_c is an approximation to the Hessian of f at the current iterate, x_+ is the next

iterate, and H_+ is the approximation to the Hessian at the next iterate, which will be used in the following step in lieu of recomputing an approximate Hessian of f at x_+ (see [2]). The vectors u and v will be chosen so that H_+ will satisfy the secant equation:

$$H_+s_c=g_+-g_c.$$

We will follow the standard convention of denoting $y_c = g_+ - g_c$. Thus the secant equation can be rewritten as

$$H_+s_c=y_c. \tag{4}$$

Initial investigations by Byrd, Schnabel, and Schultz [1] into developing a parallel secant method for unconstrained optimization focus almost entirely on performing the linear algebra calculations in parallel and using simultaneous function evaluations. While their approach is appealing for use on a vector processor and the results are good, the inherent parallelism of the problem is left untapped. One would hope for a more effective method for use on a multiprocessor.

As an alternative to the BSS method for minimizing a multivariate nonlinear function, we consider fixing the current iterate and decomposing the descent direction into its axial components, then allowing each processor to compute its part of the next iterate and update. To develop this method we must examine the inverse Broyden method.

Broyden's Method

Recall the quasi-Newton method presented above in (1)-(3). As yet, there has been no mention of how to select the vectors u and v used to perform the rank-one update to the matrix H_c . In 1965, Charles Broyden proposed the update

$$H_+ = H_c + \frac{(y_c - H_c s_c) s_c^T}{s_c^T s_c},$$

where x_+ , x_c , s_c , and y_c are as before. Since that time, the quasi-Newton methods derived from the use of similar updates have become known as Broyden methods. Note that the above update gives the least change in the affine model while remaining consistent with the secant equation, (4). If instead of using the approximate Hessian, we use an approximate inverse of the Hessian, we can formulate the inverse Broyden method.

The inverse secant method takes the form:

$$x_+ = x_c - A_c g_c \tag{5}$$

$$A_+ = A_c + u v^T \tag{6}$$

where x_c, g_c , and x_+ are as before, but A_c and A_+ are an approximation to the inverse of the Hessian at x_c and x_+ , respectively, and u and v are chosen so that A_{+} satisfies the inverse secant equation:

$$x_+ - x_c = A_+ y_c. \tag{7}$$

Note that this is exactly the secant method when $A_c =$ H_c^{-1} and $A_+ = H_+^{-1}$. If we take

$$u = s_c - A_c y_c,$$

and

$$v = \frac{A_c^T s_c}{s_c^T A_c s_c} \tag{GBU}$$

then we obtain the inverse Broyden method's update strategy

$$A_{+} = A_{c} + (s_{c} - A_{c}y_{c}) \frac{A_{c}^{T} s_{c}}{s_{c}^{T} A_{c} s_{c}}.$$
 (14)

For computational purposes, there are some simplifications in (14). First, compute

$$s_c - A_c y_c = x_+ - (x_c - A_c g_c) - A_c g_+$$

= $A_c g_+$.

Furthermore, we can rewrite

$$A_c y_c = A_c g_+ - A_c g_c = A_c g_+ + s_c.$$

Thus, we arrive at the serial implementation of the inverse Broyden method.

ALGORITHM

The Serial Secant Method Using (GBU)

Assume that x_k , A_k , and g_k are given.

- (1) Compute $s = -A_k g_k$
- (2) Compute $x_{k+1} = x_k s$
- (3) Evaluate q_{k+1}
- (4) Compute $u = A_k g_{k+1}$ (5) Compute $A_{k+1} = A_k + \frac{us^T A_k}{s^T (u+s)}$
- (6) If not converged

then increment k and go to Step (1). else set $x_* = x_k$.

Parallel Broyden's Method

Again let $f: \Omega \subset \mathbb{R}^n \to \mathbb{R}$ be a convex C^2 function with a local minimum at x_* . We know that there is a neighborhood $\mathcal{N}(x_*) \subset \Omega$ of the minimizer such

that quasi-Newton methods based on a secant approximation provide superlinear convergence on $\mathcal{N}(x_*)$. The idea is to develop a secant method which can simultaneously work in each of n orthogonal directions and still maintain superlinear convergence.

If we let $\{b_j\}$ be an orthonormal basis for \mathbb{R}^n , we can write $x_{+} = \sum_{i} \xi_{j}^{+} b_{j}$, $x_{c} = \sum_{i} \xi_{j}^{c} b_{j}$ and $g_{c} = \sum_{i} \gamma_{j}^{c} b_{j}$. Premultiplying (5) by b_i^T we get

$$\xi_i^+ = \xi_i^c - \sum_j \gamma_j^c b_i^T a_j^c \tag{8}$$

where $A_c = [a_1^c, a_2^c, \dots, a_n^c]$, so that a_j^c is the j^{th} column of A_c .

The next question is how to obtain A_+ a nonsingular approximation to $[\nabla^2 f(x_+)]^{-1}$. We seek a rank-one update of the form (6). Since this update must satisfy the inverse secant equation (7), we substitute (6) into (7) to get

$$x_+ - x_c = (A_c + uv^T)(g_+ - g_c)$$

= $A_c(g_+ - g_c) + v^T(g_+ - g_c)u$

and assuming that $v^T(g_+ - g_c) \neq 0$ we have

$$u = \frac{(x_+ - x_c) - A_c(g_+ - g_c)}{v^T(g_+ - g_c)}.$$
 (9)

Note that we are free to choose the update vector v. Then, using the u from (9), we can satisfy (7) with the rank one update in (6). We shall combine (9) and (6)

$$A_{+} = A_{c} + \frac{(x_{+} - x_{c}) - A_{c}(g_{+} - g_{c})}{v^{T}(g_{+} - g_{c})}v^{T}.$$
 (10)

Introducing the usual notation, $s = x_{+} - x_{c}$ and y = $g_{+} - g_{c}$, and assuming $v^{T}(g_{+} - g_{c}) = 1$, the matrix update becomes

$$A_{\perp} = A_c + (s - A_c y)v^T. \tag{10'}$$

Gerber and Luk [4] discuss generalized secant methods for the solution of linear systems. By considering the linear system as the gradient equation, their results can be applied to minimization of quadratic functions. They prove that the method generated by (5) and update (9) has superlinear convergence in a neighborhood of the minimizer provided that v is chosen to satisfy:

i)
$$v^T(g_+ - g_c) = 1$$

ii) $v = A_c^T w$, for some w such that
iii) $w^T(x_+ - x_c) \neq 0$

As an example, the "good" Broyden update vector

$$v = \frac{A_c^T s}{s^T A_c y} \tag{GBU}$$

satisfies the three conditions above.

Now, we wish to view $\beta_{ij}^c := b_i^T a_i^c$ as a scalar, so that (8) becomes

$$\xi_i^+ = \xi_i^c - \sum_j \gamma_j^c \beta_{ij}^c. \tag{12}$$

In order to implement this iteration on a parallel machine, we only need the *n* elements β_{ij}^c , j = 1, ..., nand not the entire matrix A_c . Similarly, we can obtain an update strategy for β_{ij}^c by

$$\beta_{ii}^{+} = \beta_{ii}^{c} + \nu_i b_i^T u \tag{13}$$

where $v = [\nu_1, \nu_2, \dots, \nu_n]$ and u is defined by (9). Applying (GBU), let us examine the term ν_i more closely. First, note that $s = x_+ - x_c = \sum (\xi_j^+ - \xi_j^c) b_j$. There-

$$s^T A_c y = \sum_{i,j} (\xi_j^+ - \xi_j^c) (\gamma_j^+ - \gamma_i^c) \beta_{ij}^c.$$

Furthermore,

$$[(A_c)^T s]_i = [s^T A_c]_i = \sum_i (\xi_j^+ - \xi_j^c) \beta_{ij}^c.$$

Putting these two formulas together yields a formula for the (GBU) v:

$$\nu_i = \frac{\sum_j (\xi_j^+ - \xi_j^c) \beta_{ij}^c}{\sum_{j,k} (\xi_j^+ - \xi_j^c) (\gamma_k^+ - \gamma_k^c) \beta_{kj}^c}.$$

If we combine all of the above into one procedure, we produce the following algorithm.

ALGORITHM

The Parallel Secant Method Using (GBU)

- (1) Set $\beta_{ij}^0 = b_j^T a_i^0$ where $A_0 = [a_1^0, \dots, a_n^0]$ for each $i, j = 1, \dots, n$. (2) Compute ξ_1^0, \dots, ξ_n^0 scalars such that $x_0 = \sum \xi_j^0 b_j$, ie. $\xi_i^0 = b_i^T x_0$.
- (3) Do in parallel $p = 1, \ldots, n$
 - (A) Set k=0.
 - (B) Evaluate $\gamma_p^k = [g(x_k)]_p$.
 - (C) vm_concat($\gamma_1^k, \ldots, \gamma_n^k$).
 - (D) Repeat

 - Repeat
 (a) Set $\xi_p^{k+1} = \xi_p^k \sum_i \gamma_i^k \beta_{ip}^k$.
 (b) vm_concat $(\xi_1^{k+1}, \dots, \xi_n^{k+1})$.
 (c) Evaluate $\gamma_p^{k+1} = [g(x_{k+1})]_p$.
 (d) vm_concat $(\gamma_1^{k+1}, \dots, \gamma_n^{k+1})$.
 (e) Set $\tau_p = \sum_j (\gamma_j^{k+1} \gamma_j^k) \beta_{jp}^k$.

 - (f) $vm_concat(\tau_1, \ldots, \tau_n)$.

(g) For
$$i = 1, ..., n$$

(i) Set $\nu_i = \frac{\sum_{j} (\xi_j^{k+1} - \xi_j^k) \beta_{ij}^k}{\sum_{j} (\xi_j^{k+1} - \xi_j^k) \tau_j}$
(ii) Set $\beta_{ip}^{k+1} = \beta_{ip}^k + \nu_i (\xi_p^{k+1} - \xi_p^k - \tau_p)$
(h) Set $k = k + 1$.

- (E) Until stopping criteria satisfied.
- (F) Return x_{k+1} to host as x_{final} .
- (4) Set $x_* = \sum \xi_i^{final} b_i$.

As a note, the vm_concat communication procedure allows data sharing over all of the nodes in the allocated subcube in logarithmic time. (See [3] for more information on the vm_concat procedure.) The stopping criteria to be used can be studied as a subject area unto itself. As an example, $||x_{k+1} - x_k|| \le \epsilon$ should be included in the stopping criteria, where $\epsilon > 0$ is the allowable error in the solution.

Numerical Results

The initial implementation of the above algorithms were done in the C programming language on the 1024 node NCUBE/Ten hypercube. In the parallel implementation, the standard basis for \mathbb{R}^n was used, and several choices for A_0 were tried. The results from the initial implementation show that the algorithm is effective on convex functions. As an example, the tables below summarize the results for the quadratic

$$f(x) = \frac{1.9}{n-1} \sum_{i=1}^{n} \sum_{j=1}^{i-1} (\xi_i - 1)(\xi_j - 1) + \sum_{i=1}^{n} (\xi_i - 1)^2$$

using the twos vector as a starting guess, i.e. $x_0 =$ $(2,2,\ldots,2)$. The secant algorithm was stopped if $||g_{+}|| < 0.000001$, $||x_{+} - x_{c}|| < 0.000001$, or the number of iterations exceeded 500. The first column d

indicates the dimension of the cube used. It is important to note that the timings listed herein include the time to load data to the nodes of the allocated cube.

For the first table, we will approximate the initial Hessian by the $n \times n$ identity matrix. This approximation has the advantage of being easily computed and positive definite.

Table 1: Quadratic, n = 32

	Using $A_0=I_n$, the identity				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	5.830	3.15e-27		
1	2	3.168	1.86e-27		
2	2	1.863	1.30e-27		
3	2	1.264	1.05e-27		
4	2	1.015	8.65e-28		
5	2	1.108	8.34e-28		

Table 1: Quadratic, n = 64

	Using $A_0 = I_n$, the identity				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	42.152	4.82e-27		
1	2	21.572	6.68e-27		
2	2	11.275	8.47e-27		
3	2	6.192	1.06e-26		
4	2	3.722	1.12e-26		
5	2	2.637	1.13e-26		
6	2	2.372	1.14e-26		

Table 1: Quadratic, n = 128

	Using $A_0 = I_n$, the identity				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	323.427	1.15e-24		
1	2	162.571	6.00e-25		
2	2	82.171	4.28e-25		
3	2	41.945	3.53e-25		
4	2	21.916	3.24e-25		
5	2	12.000	3.11e-25		
6	2	7.438	3.05e-25		
7	2	5.298	3.05e-25		

Table 1: Quadratic, n = 256

-	Using $A_0 = I_n$, the identity				
d	Iterations	Time(sec)	$f(x_*)$		
2	2	641.381	7.60e-25		
3	2	322.750	9.15e-25		
4	2	163.485	9.86e-25		
5	2	84.073	1.01e-24		
6	2	44.681	1.03e-24		
7	2	25.130	1.04e-24		
8	2	16.320	1.05e-24		

Table 1: Quadratic, n = 512

Using $A_0=I_n$, the identity				
d	Iterations	Time(sec)	$f(x_*)$	
4	2	1279.737	1.67e-23	
5	2	645.341	1.70e-23	
6	2	328.660	1.72e-23	
7	2	170.258	1.72e-23	
8	2	92.111	1.73e-23	
9	2	54.725	1.73e-23	

For Table 2, we will use a scaled version of the Identity matrix as the initial approximation of the Hessian. This approximation has the advantage of being easy to compute and is positive definite.

Table 2: Quadratic, n = 32

	Using $A_0=rac{1}{ f(x_0) }I_n$, the scaled identity				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	5.841	7.69e-31		
1	2	3.178	7.69e-31		
2	2	1.863	7.69e-31		
3	2	1.251	7.69e-31		
4	2	1.009	7.69e-31		
5	2	1.036	7.69e-31		

Table 2: Quadratic, n = 64

Using $A_0=rac{1}{ f(x_0) }I_n$, the scaled identity			
d	Iterations	Time(sec)	$f(x_*)$
0	2	42.343	0.00e+00
1	2	21.611	0.00e+00
2	2	11.319	0.00e+00
3	2	6.214	0.00e+00
4	2	3.755	0.00e+00
5	2	2.639	0.00e+00
6	2	2.380	0.00e+00

Table 2: Quadratic, n = 512

Using $A_0 = \frac{1}{ f(x_0) } I_n$, the scaled identity				
d	Iterations	Time(sec)	$f(x_*)$	
4	2	1289.277	1.23e-29	
5	2	650.342	1.23e-29	
6	2	330.784	1.23e-29	
7	2	171.689	1.23e-29	
8	2	98.123	1.23e-29	
9	2	62.651	1.23e-29	

Table 2: Quadratic, n = 128

	Using $A_0 = \frac{1}{ f(x_0) } I_n$, the scaled identity				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	325.117	0.00e+00		
1	2	163.438	0.00e+00		
2	2	82.644	0.00e+00		
3	2	42.833	0.00e+00		
4	2	22.044	0.00e+00		
5	2	12.076	0.00e+00		
6	2	7.329	0.00e+00		
7	2	5.348	0.00e+00		

Finally, we will invert a finite difference approximation of the initial Hessian. This approximation is very close to the actual inverse of the initial Hessian, unless the matrix is poorly conditioned, and hence should produce more accurate results than the other two approximations. However, performing the finite differences and the matrix inversion is slow, requiring $\mathcal{O}(n^3)$ operations, thereby increasing the start-up time on the host program. Due to the method of measuring time (only the time used on the nodes is counted), the increased start-up time is not reflected in the table.

Table 2: Quadratic, n = 256

	Using $A_0=rac{1}{ f(x_0) }I_n$, the scaled identity				
d	Iterations	Time(sec)	$f(x_*)$		
2	2	646.008	1.57e-27		
3	2	325.271	1.57e-27		
4	2	164.579	1.57e-27		
5	2	84.471	1.57e-27		
6	2	44.629	1.57e-27		
7	2	25.131	1.57e-27		
8	2	16.257	1.57e-27		

Table 3: Quadratic, n = 32

Us	Using $A_0 = abla^2 f(x_0) ^{-1}$, the inverted Hessian				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	6.060	0.00e+00		
1	2	3.233	0.00e+00		
2	2	1.923	0.00e+00		
3	2	1.270	0.00e+00		
4	2	1.018	0.00e+00		
5	2	1.039	0.00e+00		

Table 3: Quadratic, n = 64

Us	Using $A_0 = \nabla^2 f(x_0) ^{-1}$, the inverted Hessian				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	44.473	0.00e+00		
1	2	23.746	0.00e+00		
2	2	11.941	0.00e+00		
3	2	6.321	0.00e+00		
4	2	4.594	0.00e+00		
5	2	3.352	0.00e+00		
6	2	2.673	0.00e+00		

Table 3: Quadratic, n = 128

Us	Using $A_0 = abla^2 f(x_0) ^{-1}$, the inverted Hessian				
d	Iterations	Time(sec)	$f(x_*)$		
0	2	332.325	0.00e+00		
1	2	167.036	0.00e+00		
2	2	84.383	0.00e+00		
3	2	43.223	0.00e+00		
4	2	22.542	0.00e+00		
5	2	12.387	0.00e+00		
6	2	7.739	0.00e+00		
7	2	5.417	0.00e+00		

Table 3: Quadratic, n = 256

Us	$ing A_0 = \nabla^2 f($	$(x_0) ^{-1}$, the inver	ted Hessian
d	Iterations	Time(sec)	$f(x_*)$
2	2	657.612	0.00e+00
3	2	330.778	0.00e+00
4	2	167.724	0.00e+00
5	2	85.927	0.00e+00
6	2	45.366	0.00e+00
7	2	25.513	0.00e+00
8	2	16.523	0.00e+00

Table 3: Quadratic, n = 512

Us	ing $A_0 = abla^2 f($	$ x_0 ^{-1}$, the inver	rted Hessian
d	Iterations	Time(sec)	$f(x_*)$
4	2	1315.607	0.00e+00
5	2	661.409	0.00e+00
6	2	336.369	0.00e+00
7	2	174.282	0.00e+00
8	2	94.106	0.00e+00
9	2	55.777	0.00e+00

As a second example, the parallel algorithm was tested on a rather complicated exponential function

$$f(x) = \left(\sum_{i=1}^{n} x_i^2\right) \left(\frac{1}{n^2} \sum_{i=1}^{n} e^{x_i^2}\right).$$

Note that this function takes it minimum at the origin. The initial guess was the ones vector $(1,1,\ldots,1)$, and again the three initial Hessian approximations were the identity matrix, the scaled identity matrix, and the inverse of the finite difference approximation to the Hessian at the initial guess. The results for n=128 appear below in Table 4. The first table represents using the identity as the initial Hessian approximation.

Table 4: Exponential, n = 128

	Using $A_0=I$, the identity matrix					
d	Iterations	Time(sec)	$f(x_*)$			
Ö	9	1513.322	3.83e-20			
1	9	758.742	3.83e-20			
2	9	380.269	3.83e-20			
3	9	191.157	3.83e-20			
4	9	96.840	3.83e-20			
5	9	49.821	3.83e-20			
6	9	26.654	3.83e-20			
7	9	15.807	3.83e-20			

The next table represents the same problem with the same initial guess, but substituting the scaled identity for the identity as the initial Hessian approximation.

Table 4: Exponential, n = 128

Using $A_0 = f(x_0) ^{-1}I$, the scaled identity				
d	Iterations	Time(sec)	$f(x_*)$	
0	9	1511.218	6.92e-17	
1	9	757.371	6.92e-17	
2	9	379.557	6.92e-17	
3	9	190.966	6.92e-17	
4	9	97.919	6.92e-17	
5	9	51.074	6.92e-17	
6	9	28.883	6.92e-17	
7	9	16,774	6.92e-17	

The final table represents the same problem but using the computed inverse of the finite difference approximation to the Hessian at the initial guess.

Table 4: Exponential, n = 128

Us	$ing A_0 = \nabla^2 f($	$(x_0) ^{-1}$, the inver	rted Hessian
d	Iterations	Time(sec)	$f(x_*)$
0	8	1348.612	6.92e-17
ì	8	675.449	6.92e-17
2	8	346.618	6.92e-17
3	8	168.983	6.92e-17
4	8	85.201	6.92e-17
5	8	43.370	6.92e-17
6	8	22.567	6.92e-17
7	8	12.407	6.92e-17

From the timing results, an efficiency rating can be determined for each run of the algorithm. This rating is a measure of how much parallelism can be exploited by using more nodes on the problem. The higher the efficiency rating, the more concurrent work is being performed. Large amounts of communication time lowers the efficiency rating considerably. The efficiency is taken to be the execution time of the algorithm relative to the number of nodes used, ie.

efficiency =
$$\frac{\text{time}_1}{p * \text{time}_p}$$

where $time_1$ is the execution time of the algorithm on 1 node, and $time_p$ is the execution time of the algorithm using p nodes. Note that the start-up cost incurred by the host processor (reading x_0 , evaluating g_0 , and

approximating A_0) is not counted, but that the startup cost incurred by a node (receiving x_0, g_0 and the appropriate section of A_0) is counted in the execution time. The efficiencies presented in the table below are surprisingly very high when the gradient is expensive to evaluate in relation to communication time. This is the case for larger values of n. Due to memory restrictions on the nodes (roughly 400 kilobytes of user space is available), running the n = 256 quadratic was not possible on a cube of dimension 0 or 1. Rather than extrapolate the timings for the n = 256 problem on 1 or 2 nodes, the efficiency ratings for that problem have been omitted. Similarly, there are no efficiency results for the n = 512 problem. The rest of the efficiency ratings for the quadratic are listed in Table 5 below. Missing values are indicated by two asterisks in the field. (These are cases when the number of nodes used would exceed the dimension of the problem forcing idle time, or unnecessary replication of work.)

Table 5: Efficiency, Quadratic

Us	$ing A_0 = \nabla$	$ x_0 ^{-1}$, the	inverted Hessian
d	32	64	128
0	1.0000	1.0000	1.0000
1	0.9370	0.9364	0.9948
2	0.7878	0.9311	0.9846
3	0.5964	0.8794	0.9611
4	0.3719	0.6049	0.9214
5	0.1821	0.4145	0.8383
6	**	0.2599	0.6709
7	**	**	0.4792

The results of the efficiencies for quadratic function above with n=128 are summarized in the three figures below. The similarities of the efficiencies between any two of the three cases precludes their combined presentation in one plot.

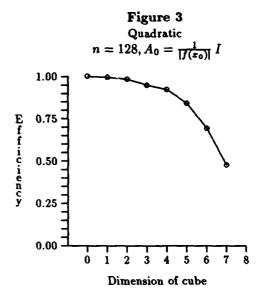
Figure 1
Quadratic $n = 128, A_0 = (\nabla^2 f(x_0))^{-1}$ E 0.75

f

i 0.50

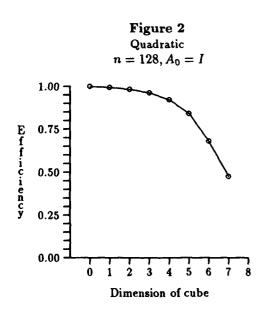
o 1 2 3 4 5 6 7 8

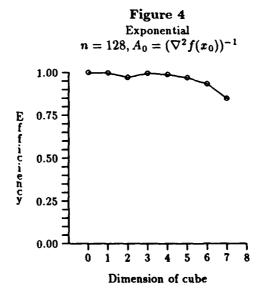
Dimension of cube

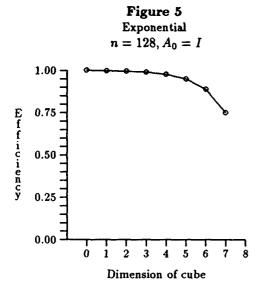


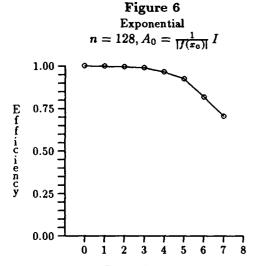
Finally, for comparative purposes, the efficiencies for the exponential function with n=128 are shown in the next three figures (again, one for each of the three methods for choosing the initial Hessian approximation). Note that these three plots appear very similar to the previous plots. In fact, the plots from all of the test cases used to test the algorithm take the same form. The small dip in Figure 4 can be attributed to the inaccuracy of taking measurements while multiple jobs were running on the NCUBE.

The high efficiencies correspond to ratings of between 80.0 and 90.0 kiloflops on each of the 128 nodes. This agrees with the findings of Gustafson, Montry, and Benner [5] that sustained performance is between 70 and 130 megaflops for the NCUBE/Ten (68.4 to 127.0 kiloflops on a single node, double precision).









Dimension of cube
There were several other test functions used in evaluating the performance of the parallel secant method, but those that did not diverge did not provide as clear an illustration of the performance characteristics of the method. Hence, they have been omitted.

Conclusions

These results suggest that the method is efficient in the minimization of a multivariate nonlinear function, and that while the number of iterations required for a solution, and the time required to obtain that solution, are dependent upon the initial approximation for the inverse of the Hessian, the extra accuracy and speed gained by using the finite difference approach does not justify the additional start-up penalty incurred. The method works well for convex functions, but is subject to the usual limitations of an inverse secant method: bad initial approximations or noisy functions can cause the iterates to travere infeasible regions, or to diverge. force convigence in these cases, trust regions, line earches, or backtracking techniques must be added. By adding a "corrective" technique, the parallel secant method will provide a very good alternative to other minimization methods, particularly when the problem size increases, and function evaluation becomes more costly.

References

- [1] Byrd, R., Schnabel, R., and Shultz, G. "Parallel Quasi-Newton Methods for Unconstrained Optimization," Mathematical Programming 42 (1988) 273-306.
- [2] Dennis, J., and Schnabel, R. Numerical Methods for Unconstrained Optimization and Nonlinear Equations. Prentice-Hall, Englewood Cliffs, NJ (1983).
- [3] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. and Walker, D. Solving Problems on Concurrent Computers, Vol I. Prentice-Hall, Englewood Cliffs, NJ (1988).
- [4] Gerber, R., and Luk, F. "A Generalized Broyden's Method for Solving Simultaneous Linear Equations," Technical Report TR-80-438, Department of Computer Science, Cornell University, Ithaca, NY (1980).
- [5] Gustafson, J. L., Montry, G. R., and Benner, R. E. "Development of Parallel Methods for a 1024-Processor Hypercube," SIAM Journal on Scientific and Statistical Computing. Volume 9, Number 4, 1988.

Acknowledgments

I would like to thank the entire staff at the University of South Carolina's Parallel Supercomputer Initiative for their help and advice in this research project. In particular, thanks go to Chuck Baldwin for providing technical assistance and the communications routines mentioned herein. Especially, I would like to thank George Johnson for providing ideas, suggestions and direction.

Parallel Nonlinear Optimization

Ron Daniel Jr.
Oklahoma State University
School of Electrical and Computer Engineering
202 Engineering South
Stillwater, OK 74078

Abstract

This paper describes the implementation of a parallel Levenberg-Marquardt algorithm on an iPSC/2. The Levenberg-Marquardt algorithm is a standard technique for non-linear least-squares optimization. For a problem with D data points and P parameters to be estimated, each iteration requires that the objective function and its P partials be evaluated at all D data points, using the current parameter estimates. Each iteration also requires the solution of a PxP linear system to obtain the next set of parameter estimates. A simple data-parallel decomposition is used where the data is evenly distributed across the nodes to parallelize the evaluations of the objective function and its partial derivatives. The performance of the method is characterized versus the number of nodes, the number of data points, and the number of parameters in the objective function. Further enhancements are also discussed.

Introduction

Many problems can be cast as the search for a set of parameters that minimize (or maximize) some function. Such a problem is known as a minimization or optimization problem. The function to be minimized is known as the objective function. Classes of algorithms exist for cases where the objective function is linear vs. non-linear, univariate vs. multivariate, and where its derivatives with respect to the adjustable parameters are known vs. unknown.

The Levenberg-Marquardt (LM) algorithm [1,2] is a standard technique for non-linear least-squares, multivariate optimization when the partial derivatives of the objective function are known and are not too inconvenient to compute. For a problem with D data points and P parameters to be estimated, each iteration requires that the objective function and its P partials be evaluated at all D data points using the current parameter estimates. Each iteration also requires the solution of a PxP linear system to obtain the next set of parameter estimates. P is almost always much less than D.

This paper describes the implementation of a parallel LM algorithm on an iPSC/2 SX (Weitek FPUs). The data is evenly distributed across the nodes to parallelize the evaluations of the objective function and its partial derivatives. Since the computation of the objective function and its partials at one data point is

independent of the computations at the other data points, this phase of the problem is perfectly parallel. Currently, the linear system solution is carried out on one node.

Levenberg-Marquardt Technique

In this section we will discuss the LM algorithm in enough detail to see how it is parallelized. A full derivation can be found in [1], [2], or almost any book on non-linear optimization. The presentation below follows [2].

The goal of the LM algorithm is to minimize the objective function:

$$\chi^{2}(\mathbf{p}) = \sum_{i=1}^{N} (y_{i} - y(x_{i}, \mathbf{p}))^{2}$$
 (1)

by iteratively adjusting the vector of parameters, \mathbf{p} . The \mathbf{x}_i are the independent variables of the data, the \mathbf{y}_i are the measured dependent values, and the function $\mathbf{y}(\mathbf{x}_i, \mathbf{p})$ predicts \mathbf{y}_i given \mathbf{x}_i and the current parameter estimates.

The components of the gradient vector and second derivative matrix of χ^2 are:

$$\frac{\partial \chi^2}{\partial p_k} = -2 \sum_{i=1}^{N} (y_i - y(x_i, p)) \frac{\partial y(x_i, p)}{\partial p_k}$$
 (2)

and
$$\frac{\partial^{2}\chi^{2}}{\partial p_{k}\partial p_{l}} = 2\sum_{i=1}^{N} \left[\frac{\partial y(x_{i},p)}{\partial p_{k}} \frac{\partial y(x_{i},p)}{\partial p_{l}} \right]$$
(3)
$$- (y_{i} - y(x_{i},p)) \frac{\partial^{2}y(x_{i},p)}{\partial p_{k}\partial p_{l}}$$

We will remove the factors of two by defining the components of the vector **b** and matrix **A** as

$$\mathbf{b} = [\mathbf{b}_k] = \left[\frac{-1}{2} \frac{\partial \chi^2}{\partial \mathbf{p}_k}\right], \ \mathbf{A} = [\mathbf{a}_k] = \left[\frac{1}{2} \frac{\partial \chi^2}{\partial \mathbf{p}_k}\right]$$
(4.5)

If χ^2 can be accurately approximated by a quadratic surface, then the correction ∂p (which when added to the current parameter estimates p gives the parameters p_{min}

that minimize χ^2) can be found in a single step by solving the linear system:

$$\mathbf{A} \, \partial \mathbf{p} = \mathbf{b} \tag{6}$$

Note that A is a function of both the first and second derivatives of χ^2 . The second derivatives will be destabilizing when the quadratic is not a good approximation. We will therefore neglect these terms and redefine A as:

$$\mathbf{A} = [\mathbf{a_{kl}}] = \left[\begin{array}{cc} \mathbf{N} \\ \sum_{i=1}^{N} & \left(\frac{\partial \mathbf{y}(\mathbf{x_i, p})}{\partial \mathbf{p_k}} & \frac{\partial \mathbf{y}(\mathbf{x_i, p})}{\partial \mathbf{p_l}} \end{array} \right) \right]$$
(7)

This change improves stability and lessens the computational complexity without seriously degrading performance of the method.

If the quadratic approximation is a bad assumption, the step of (6) will probably cause χ^2 to increase rather than decrease. In this case, about the best we can do is take a step down the gradient:

$$\partial \mathbf{p} = \mathbf{t} \, \mathbf{b},$$
 (8)

where t is a constant that sets the size of the step. Even if the quadratic model is only a fair approximation, A provides some information about the size of t. Marquardt defined a new matrix A':

$$A' = A + \lambda I \tag{9}$$

When λ is small, our update is essentially that of (6). When λ is large, A' becomes diagonally dominant, and our step approaches an infinitesimal step down the gradient. As long as our steps are succeeding, we can assume the quadratic approximation to χ^2 is accurate, and can decrease λ to achieve faster convergence. If a proposed step fails, we increase λ and try again. This is expressed below in Algorithm 1.

Algorithm 1 Serial Levenberg-Marquardt Technique

```
    0 input initial p, data x, y, set λ = 0.001
    1 compute χ²(p), b, A
    2 if || b || < tol or iteration limit reached, done</li>
    3 solve (A + λI) ∂p = b for ∂p
    4 compute χ²(p + ∂p), b<sub>tmp</sub>, A<sub>tmp</sub>
    5 if χ²(p + ∂p) ≥ χ²(p)
        λ = λ * 10
    else
        λ = λ / 10, p = p + ∂p,
        b = b<sub>tmp</sub>, A = A<sub>tmp</sub>
    6 go to 2
```

Parallel Decomposition

The LM algorithm is quite easy to parallelize using a data-parallel decomposition. At the beginning of the process, the x and y data is evenly distributed among the nodes. At the start of each iteration, the current parameter estimates are broadcast to all the nodes. Each node then computes χ^2 , b, and A for its portion of the data. These are summed and node 0 receives the total χ^2 , b, and A. It solves the linear system to determine the new vector of parameter estimates. This is expressed below in Algorithm 2.

Algorithm 2 Parallel Levenberg-Marquardt Technique

```
broadcast initial p to all nodes,
set λ = 0.001

1 compute χ²(p), b, A in parallel
2 if || b || < tol or iteration limit reached,
done
3 solve (A + λI) ∂p = b for ∂p
4 broadcast p + ∂p to all nodes
5 compute χ²(p + ∂p), b<sub>tmp</sub>, A<sub>tmp</sub> in parallel
6 if χ²(p + ∂p) ≥ χ²(p)
λ = λ * 10
else
λ = λ/10
p = p + ∂p
b = b<sub>tmp</sub>, A = A<sub>tmp</sub>
```

0 distribute x, y data evenly among nodes,

It would also be possible to parallelize the linear system solution in step 3. The advantages and disadvantages of this are discussed in the conclusions section.

Once the data has been distributed, the only communication that occurs is the broadcast of the new parameter estimates and the summation of χ^2 , b, and A. This communication does not increase as the number of data points increases. The communication does increase as the log of the number of nodes and the square of the number of parameters.

Performance

Several factors make it difficult to characterize the performance of the LM algorithm. First, it is an iterative algorithm, so the possibility exists that as more data points are added, enough additional information is obtained to reduce the number of iterations needed to converge to a set of parameter estimates. For this reason we report per-iteration times. The number of iterations will not be affected by the number of nodes used.

Another problem is that the measures of speedup and efficiency are influenced by the complexity of the objective function calculations relative to the linear system solution. For example, consider the case where the evaluations of the objective and its partials take very little time. The computation time will be dominated by that of the linear system solution, and will show no dependence upon the number of data points. Further, performance would be expected to degrade as additional nodes are added, due to the increased communication costs. The opposite case, where the evaluation of the objective function and its partials is quite lengthy, would exhibit almost perfectly linear speedup. Neither of these results would accurately predict the performance for objective functions of moderate complexity. Our objective function should also allow us to characterize performance as the number of parameters increases. To accommodate both these requirements, we will use an objective function that models the input data as the sum of K Gaussians. By increasing K, we can increase the computational complexity of the objective function. Each Gaussian is characterized by three parameters, its location lk, its amplitude ak, and its spread, sk. The objective function is then

$$\chi^2 = \sum_{i=1}^{D} (y_i - y(x_i, p))^2$$
 (10)

where x_i is the independent variable(s) of the i'th data point, y_i is the dependent value, and $y(x_i,p)$ is the function value predicted from the current set of parameters, p:

$$y(x_i, p) = \sum_{k=1}^{K} a_k \exp\left(\left(-\frac{x_i - l_k}{s_k}\right)^2\right)$$
 (11)

K is the number of Gaussians, thus the number of parameters in the model is P = 3K. An example of the input data and the underlying Gaussians is shown below in figure 1 for the case K=3.

Figure 2a shows the time for a single iteration, including the linear system solution, versus the number of nodes and number of data points for the case of 4 Gaussians (12 parameters). Figure 2b shows the time for a single iteration versus the number of parameters and number of nodes for the case of 5000 data points. These times do not include the time to read the data into the nodes. The decision not to include I/O times was made because I/O times are highly dependent upon the presence or absence of the iPSC/2 Concurrent I/O hardware.

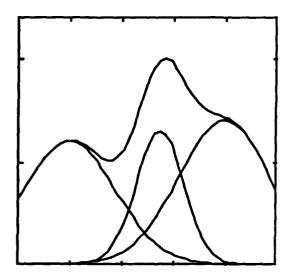


Figure 1: Input Data and Underlying Gaussians

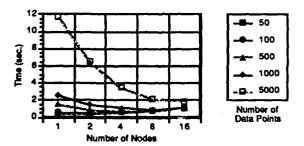


Figure 2a: Time / Iteration vs. Number of Nodes and Number of Data Points, 12 Parameters

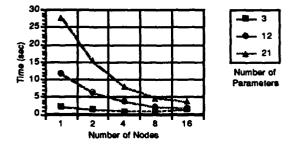


Figure 2b: Time / Iteration vs. Number of Nodes and Number of Parameters, 5000 Data Points

Conclusions

Figure 2a shows that for small data sets, the increase in communication as the number of nodes increases is not made up for by the reduction in time by evaluating the objective function and its partials in parallel. In fact, we see a slowdown as the number of nodes increases. As can be expected, this problem is reduced when the data sets become larger, as shown in

figure 2a. The same problem is seen for small numbers of parameters, as shown in figure 2b.

This problem has two causes. Most of the nodes are sitting idle while one node performs the linear system solution. This idle time, combined with the increased communication mentioned above, leads to the poor speedups noted as the number of nodes increases.

The efficiency of the algorithm could be increased if we could speed the linear system solution, thus eliminating the time where all nodes but one are idle. The obvious solution is to parallelize the linear system solution, which is the next step we will try. However, its benefits are doubtful. For small numbers of parameters more overhead will be incurred than would be compensated for by parallel execution. In other words, it will merely aggravate the current problem. This will be alleviated as the number of parameters increases, but reasonable efficiencies will probably not be reached until there are about 10 times as many parameters as nodes [3]. However, for problems of that size it is time to see if an algorithm other than LM can be used. LM is an excellent general purpose algorithm. but for such large numbers of parameters it is better to find an algorithm that exploits special, problemdependent characteristics of χ^2 , b, or A.

References

- [1] Scales, L.E. (1985) Introduction to Non-Linear Optimization, Springer-Verlag, New York.
- [2] Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T. (1988) *Numerical Recipes in C*, Cambridge University Press, Cambridge.
- [3] Juszczak, J.W. and van de Geijn, R.A. (1989) An Experiment in Coding Portable Parallel Matrix Algorithms. In Proceedings of the Fourth Conf. on Hypercubes, Concurrent Computers, and Applications (HCCA4), Monterey, CA, April, 675-680.

Parallelizing Multiple Linear Regression for Speed and Redundancy: An Empirical Study¹

Mingxian Xu John J. Miller Edward J. Wegman

Center for Computational Statistics George Mason University Fairfax, VA 22030

ABSTRACT

The purpose of this paper is to present a parallel implementation of multiple linear regression. We discuss the multiple linear regression model. Traditionally parallelism has been used for either speed-up or redundancy (hence reliability). With stochastic data, by clever parsing and algorithm development, it is possible to achieve both speed and reliability enhancement. We demonstrate this with multiple linear regression. Other examples include kernel estimation and bootstrapping.

1. Introduction

Contemporary statistical computations often focus the analysis of massive data sets with complex algorithms. Consequently, efforts to speed up the calculations are extremely important even with the impressive computational power available today. Parallel computation techniques are an important technology that may be used to achieve speedup and, indeed, are likely to become even more significant as the physical limits of conventional serial architectures are reached. Historically in the 1960s and early 1970s, parallelism was also used in the design of both hardware and software to enhance the reliability of systems through redundancy. In such a design,

components (either hardware or software) run in parallel performing the same task. The parallel processors each process the same data, with a voting procedure used to determine the reported outcome of the computation. The object of the redundancy in this case is fault tolerance. Of course, this type of parallelism leads to no inherent speed-up in the computations.

One may use parallelism in achieving speed-up by sending different data to the different processors. This can result in substantial speed-up. depending communication overhead and the details of the implementation of parallelism. However, in this mode of operation there is no mechanism for achieving fault detection. For example, the decomposition of an integral and assignment of portions of that integral to processors in a numerical quadrature algorithm is an illustration of this sort of parallelism. It would usually be impossible to know whether one processor returned an incorrect value for its portion of the integral. An interesting question then is whether or not there are situations where we can use parallelism for speed-up and still maintain some of the properties of redundancy for our reliability checks. In fact, the thesis of this paper is that this is possible in some situations, as will be described below.

¹This research was supported by the Army Research Office under contract number DAAL03-87-K-0087, by the Office of Naval Research under contract number N00014-J-89-1807 and by the Virginia Center for Innovative Technology.

In a setting in which data may be be generated from assumed to some probabilistically homogeneous structure, we are suggesting the use of statistical hypothesis tests in place of voting procedures to compare results from different nodes. We assign data to nodes by parsing in an appropriate manner. As a first step, we assume that the data may be parsed into random samples. Since we began with stochastically homogeneous data and parsed it into random samples, the only variation in the output that we should expect to see from the different nodes is stochastic variation. Hence, we can use statistical tests to check the results for deviations from homogeneity. These tests yield a stochastic measure of redundancy for our parallel implementation. We can, thus, use the tests for fault detection in either of the node hardware or software.

To illustrate these ideas, we have selected multiple linear regression as an application. We parallelize the computations for multiple regression and then use the results from each node as a part of a homogeneity check. We have selected multiple regression because the procedure is well understood, the computations are straightforward, and the statistical tests of homogeneity are easily developed. Subsequently, we develop other applications such as kernel density estimation bootstrapping.

Our implementation of the above described parallel techniques will take place using an Intel iPSC/2 (referred to in this paper as the hypercube). Our hypercube is configured with 16 nodes, each of which has a 32-bit 80386 CPU, an 80387 math co-processor, a direct routing module for communication via message passing and an additional vector pipeline coprocessor. In addition to the 16 nodes, there is also a host node (referred to by Intel as the system resource manager), which "directs" the activity of the other nodes. The hypercube has a distributed, message-passing architecture. Data passes through the host to the nodes and the results are gathered from the nodes back to the host. Given the messagepassing nature of the architecture, communication overhead typically plays a

significant part in the overall effectiveness of any algorithm. In general, computational problems which require comparatively little internode communication are the most effective ones on the message passing architectures. Bootstrapping and kernel smoothing operations are examples of computationally intensive tasks which fall into this category. While multiple linear regression is comparatively communications intensive, it does admit a very effective parallel implementation and, moreover, elegantly illustrates our point that we felt it was quite worth developing. Also it allows us to investigate the effect of varying communication packet size on the potential speed-up.

2. The Multiple Linear Regression Model

We often have the need to study a system in which the changes in several variables may effect the dependent variable. We may know or be willing to assume that the model is expressed as a linear model or we may use a linear model as an approximation to some unknown, more complex model. In either case, least squares estimation yields a computational technique generally known as regression. We distribute the computations necessary for multiple linear regression over several node We then use statistical tests for processors. as a redundancy check homogeneity hardware and software faults. The tests used in this discussion depend on the assumption of normally distributed residuals for their complete validity although, of course, their nonparametric analogues may also be used. Because our use of these normal tests is as a descriptive statistic to indicate severe deviations from homogeneity, we are not extremely concerned whether the assumption of normality is met exactly or not.

The mathematical model for multiple linear regression can be expressed as follows:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + e_i,$$

 $i = 1, 2, ..., n,$

or in matrix formulation:

$$y = \frac{1}{2}\beta_0 + X\beta_1 + \varepsilon,$$

where y is an (n x 1) vector of observations, $\underline{1}$ is an (n x 1) vector of ones, β_0 is an unknown parameter, X is an (n x p) matrix of nonstochastic variables, β_1 is a (p x 1) vector of unknown parameters, and e is an (n x 1) vector of random errors. The traditional assumptions are E(e) = 0 and $Cov(e) = \sigma^2 I$. Thus $E(y) = X\beta$ and $Cov(y) = \sigma^2 I$. It can be shown that the least squares estimates of β_0 and β_1 , β_0 and β_1 , may be obtained as follows:

$$\hat{\beta}_1 = (\widetilde{X}'\widetilde{X})^{-1}\widetilde{X}'\widetilde{y}, \, \hat{\beta}_0 = \overline{y} - \overline{x}'\hat{\beta}_1, \quad (1)$$

where $\bar{\mathbf{x}} = \overset{\mathbf{X}'1}{\widetilde{\mathbf{x}}'}$ is the vector of column means, $\overset{\mathbf{X}}{\widetilde{\mathbf{x}}} = \overset{\mathbf{X}}{\mathbf{x}} - \overset{\mathbf{X}}{\widetilde{\mathbf{x}}} \overset{\mathbf{Y}}{\widetilde{\mathbf{x}}}$ is the centered $\overset{\mathbf{X}}{\mathbf{x}}$ matrix so that $\overset{\mathbf{X}}{\widetilde{\mathbf{x}}} : \overset{\mathbf{X}}{\mathbf{x}} : \overset{\mathbf{X}}{\mathbf{x}}$

To implement multiple linear regression in a parallel fashion, we partition the whole set of the observations and variables into m subsets of close to equal size. We then send each of these subsets to one of the m nodes. We denote data sent to, computed at, or received from node k by adding a subscript (k) to the item. Thus we send to node k: $y_{(k)}$ and $x_{(k)}$ of $x_{(k)}$ rows each. We compute at node k: $x_{(k)}$, $x_{(k)}$, x

and $SSE_{(k)}$. We note two things at this point: 1) We center at each node. (We use a one pass recursive centering algorithm for speed and accuracy.) 2) We do not compute the slopes and intercepts at each node, although we could if we wished. We are not going to use the node estimates. We merely wish to use the information returned from the nodes to i) Compute the least squares estimates for all the data and ii) Assess homogeneity of the results for fault checking.

In order to proceed with our homogeneity checks, we define three potential models for our data. Model 0 is the nominal model. For each node partition of the data, we

assume that the slope vector β_1 , and the intercept β_0 are the same. For Model 1, we assume that the node partitions have the same slope vector, but different intercepts. For Model 2, we assume that the node partitions have both different slope vectors and different intercepts. In matrix terms, the three models are given by:

Model 0:
$$y_{(k)} = 1\beta_0 + X_{(k)}\beta_1 + \varepsilon_{(k)}$$
.

Model 1:
$$y_{(k)} = \frac{1}{2}\beta_{0(k)} + X_{(k)}\beta_1 + e_{(k)}$$
.

Model 2:

$$y_{(k)} = 1 \beta_{0(k)} + X_{(k)} \beta y y_{1(k)} + e_{(k)}$$

After aggregating at the host the information computed at the nodes, we proceed to compute a Sum of Squared Errors for each of the three models as follows:

Model 2:
$$SSE_2 = \sum_{k=1}^{m} SSE_{(k)}.$$

Model 1:
$$(\widetilde{\mathbf{y}}'\widetilde{\mathbf{y}})_{(1)} = \sum_{k=1}^{m} (\widetilde{\mathbf{y}}'_{(k)}\widetilde{\mathbf{y}}_{(k)}),$$

$$(\widetilde{\mathfrak{Z}}'\widetilde{\mathfrak{Z}})_{(1)} = \sum_{k=1}^{m} (\widetilde{\mathfrak{Z}}'_{(k)}\widetilde{\mathfrak{Z}}_{(k)})$$

$$(\widetilde{\boldsymbol{\mathfrak{X}}}'\widetilde{\boldsymbol{\mathfrak{X}}})_{(1)} = \textstyle\sum\limits_{k=1}^{m} (\widetilde{\boldsymbol{\mathfrak{X}}}'_{(k)}\widetilde{\boldsymbol{\mathfrak{Y}}}_{(k)})$$

$$SSE_{1} = (\widetilde{\chi}'\widetilde{\chi})_{(1)} - (\widetilde{\chi}'\widetilde{\chi})'_{(1)}(\widetilde{\chi}'\widetilde{\chi})^{-1}_{(1)}(\widetilde{\chi}'\widetilde{\chi})^{-1}_{(1)}$$

Model 0:

$$\begin{split} \mathbf{n} &= \sum_{k=1}^{\mathbf{m}} \mathbf{n}_k, \, \bar{\mathbf{x}} = \sum_{k=1}^{\mathbf{m}} \mathbf{n}_k \bar{\mathbf{x}}_{(k)} / \mathbf{n}, \, \bar{\mathbf{y}} = \sum_{k=1}^{\mathbf{m}} \mathbf{n}_k \bar{\mathbf{y}}_{(k)} / \mathbf{n}, \\ (\widetilde{\mathbf{y}}' \widetilde{\mathbf{y}})_{(0)} &= (\widetilde{\mathbf{y}}' \widetilde{\mathbf{y}})_{(1)} + \sum_{k=1}^{\mathbf{m}} \mathbf{n}_k (\bar{\mathbf{y}}_{(k)} - \bar{\bar{\mathbf{y}}})^2, \end{split}$$

$$(\widetilde{\mathbf{X}}'\widetilde{\mathbf{y}})_{(0)} = (\widetilde{\mathbf{X}}'\widetilde{\mathbf{y}})_{(1)} + \sum_{k=1}^{m} n_{k}(\bar{\mathbf{x}}_{(k)} - \bar{\bar{\mathbf{x}}})(\bar{\mathbf{y}}_{(k)} - \bar{\bar{\mathbf{y}}})$$

$$(\widetilde{\chi}'\widetilde{\chi})_{(0)} = (\widetilde{\chi}'\widetilde{\chi})_{(1)} + \sum_{k=1}^m \, \mathrm{n}_k (\bar{\chi}_{(k)} \, - \, \bar{\bar{\chi}}) (\bar{\chi}_{(k)} \, - \, \bar{\bar{\chi}})'$$

$$SSE_0 = (\widetilde{\mathbf{y}}'\widetilde{\mathbf{y}})_{(0)} - (\widetilde{\mathbf{x}}'\widetilde{\mathbf{y}})'_{(0)}(\widetilde{\mathbf{x}}'\widetilde{\mathbf{x}})^{-1}_{(0)}(\widetilde{\mathbf{x}}'\widetilde{\mathbf{y}})_{(0)}$$

We calculate the regression solutions using equation (1) with the summary statistics computed for Model 0. The degrees of freedom associated with the Error Sums of Squares are given by $df_0 = n - p - 1$, $df_1 = n - p - m$, $df_2 = n - mp - m$.

We may now calculate two test statistics. We may test for Total Homogeneity (which is the true redundancy test for fault checking) and for Homogeneity of Slopes Only (which we have included simply because it is so easy to do and might provide some detail about what went wrong if something did). The test for Total Homogeneity uses the statistics: $SS_{(2,0)} = SSE_0 - SSE_2$, $df_{(2,0)} =$

$$df_0 - df_2 = (m-1)(p+1), MS_{(2,0)} = SS_{(2,0)}/df_{(2,0)},$$

$$F_{(2,0)} = MS_{(2,0)}/MSE_2$$
, where $MSE_2 = SSE_2/df_2$.

The test for Homogeneity of Slopes Only uses the statistics: $SS_{(2,1)} = SSE_1 - SSE_2$, $df_{(2,1)} = df_1 - df_2$

=
$$(m-1)(p+1)$$
, $MS_{(2,1)} = SS_{(2,1)}/df_{(2,1)}$, $F_{(2,1)}$

 ${\rm MS}_{(2,1)}/{\rm MSE}_2$, where ${\rm MSE}_2={\rm SSE}_2/{\rm df}_2$. If heterogeneity is detected, then further tests may be made to isolate the nodes with different and presumed faulty results. We note at this point that we set the significance level for our homogeneity test very small. This is because we want a very small false alarm rate. We only want to detect egregious deviations from homogeneity, as might be caused by a hardware or software failure.

The methodology described above is designed to isolate potentially catastrophic failures in the node hardware or software.

However, it could also be used to affect a speedup of other kinds of homogeneity checks on For example, suppose that instead of parsing the data to allocate it to nodes in a manner which creates random samples, we allocated the data to correspond to some meaningful partition of the data such as orthants of the X space. The parallel algorithm described above would then yield a speed-up of this homogeneity check, but would no longer have any fault detection capability. However, a simple modification whereby we split each partition into two or more subpartitions via random sampling would still give a homogeneity check using straightforward extensions of the above methodology.

3. The Timing Results

The timing study is designed to measure the effectiveness of the parallel scheme described above, and, in particular, to measure the effect of changing the size of the communications packets sent between the host and the nodes. We began by generating data files of similar data. We did this by taking an original data set with six independent variables and then generating data sets of arbitrary size. We then matched the covariance structure of the rows of the X matrix with that of the original problem, made the regression coefficients the same as in the original problem, and matched the variability of the generated residuals with those of the original problem. Hence, regardless of the size of the test data set, we could be assured that it stochastically agreed with the original data set. In this sense, our test data sets were comparable. The sizes selected for this part of the study were n = 8000 and 16000observations. We also used various numbers of nodes, so that the speed-up from parallelizing could be determined.

The study also measured the effect of differing sizes of communication packets sent from the host to the nodes. The sizes used in this study were 125, 250, and 500 observations per node. Since we used a "broadcast" transmission of the data for all nodes, with each node picking its data out of the message, the size of packet transmitted also depends on the

number of nodes. The size of the actual transmitted packet is number of nodes times package size. It might appear that one should automatically choose the largest possible size of packet in order to minimize the effect of communications start up overhead. However, this could lead to nodes remaining idle while transmission is taking place. Hence, it may be more effective to use smaller packets, so that the nodes may continue doing productive work. We used several sizes to examine the effect of package size on overall efficiency.

We measured at each node the overall time for the program, the time waiting for the host to read data, the computation time, and data transmission time (both sending and receiving). Our measure of effective time for the computations is the maximum over nodes of overall time minus time waiting for the host to Hence, the computation time read the data. and the communications overhead time for the hypercube are included in our time measure, but the time for the host to initially read in the data is not included. The speed-up for any given number of nodes for a particular configuration is given by the ratio of the time for one node divided by the time measure for that number of nodes. Times were measured by an internal clock subroutine on the hypercube and are given in milliseconds. The results of our simulations are given in Table 1. Each number is the average for two runs.

We observe from Table 1 that the effective times indeed decrease as we add more nodes. We also note that the speed-up is not linear in the number of nodes. The speed-ups achieved for the six rows of Table 1 (from one node to sixteen) are respectively: 12.65, 13.55, 11.79, 13.16, 10.43, and 12.31. The speed-ups are greater for the larger data set and are greater for package size 125 observations per node than for the larger packages. The reason that speed-up is not perfectly linear is that communications overhead increases as the number of nodes increases. However, as might be expected for perfectly parallel computations as we have here, the computation time indeed decreases as the reciprocal of the number of nodes. In fact, a regression of computation time

(again the maximum over nodes) versus sample size divided by number of nodes yields an R² of 0.999978. The communication overhead prevents the speed-up from achieving perfect linear speed-up.

We also made some additional runs with larger sample sizes to explore the limiting behavior of the speed-up. We wished to observe where the asymptote, if any was with respect to increased speed-up and sample size. Hence, we made additional runs with one and sixteen nodes for each package size for sample sizes 32000, 64000, and 128000. The results of these runs and some information from Table 1 are presented in Table 2.

As may be seen from Table 2, the speed-up appears to have an asymptotic value of approximately 13.8 for sixteen nodes. This seems to be the case regardless of the package size. Nevertheless, for any given sample size, the smaller package size gives smaller effective times and larger speed-up. Hence, we observe the phenomenon that the desired efficiency of the larger package size (namely, the lesser number of times the communication startup overhead is involved) is overcome by the fact that the nodes sit idle waiting for data to arrive with the larger package sizes.

We make one final remark with regard to effective time. If the time at the host for reading in the data is included, the time to read in the data overwhelms the computations for this problem. We conceive of a situation in which the data are acquired in some automated mode which can bypass the reading step that we This is reasonable, since the fault did here. checking feature described above would be critical in a situation where the data arrived in huge amounts and was processed in an automated fashion. The effective time as we have measured it gives a fair reading of the speed-up from the parallel implementation of All communication overhead is regression. included except the reading of the data. This is a commonly applied method for measuring speed-up.

4. A Short Example Using Kernel Estimation and Bootstrapping

We note that the increase in efficiency (speed-up) is small for regression calculations when input time is included. Therefore, we have also used the Hypercube to generate speedup for a more computationally intensive statistical procedure, namely Kernel Estimation with Bootstrapping. Density estimation is motivated as follows. Suppose that a set of observed data is assumed to be a sample from an unknown probability density function. We wish to construct an estimate of the underlying density from the observed data. Kernel density estimation is a nonparametric technique used to accomplish this estimation.

Suppose that the underlying density is f(x). The kernel density estimate $\hat{f}(x)$ is defined by:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K(\frac{x - X_i}{h}),$$

where n is the sample size, h is the width of the window of the kernel, the X_i are the observed data, and $K(\cdot)$ is the kernel function. $K(\cdot)$ satisfies the following conditions: $\int K(x) dx = 1$, $\int x K(x) dx = 0$, $\int K^2(x) dx < \infty$, $\int |x|^p K(x) dx < \infty$, for some p. The limits of integration are selected appropriately for the particular kernel function. Use of a kernel density estimate is similar to use of a weighted average to estimate a parameter.

We performed a small timing study of parallelizing kernel density estimation. We used 10000 data points and used four kernel functions to obtain four different kernel density estimates. The total processing time to accomplish this task is the outcome measure. All overhead time (including time reading the data) is accounted for in the measure.

Again we use the maximum of the node times as our measure of the node processing time. Time is in milliseconds and each time is for one run. The results for 1, 2, 4, 8, and 16 nodes were respectively: 8,292,879; 4,143,741; 2,082,504; 1,055,863; and 545,479. The speedup from one to sixteen nodes was 15.20. The parallelism achieves close to perfect linear speed-

up in the number of nodes. The communication time is extremely small compared to the large amount of computation time for this application.

Bootstrapping is another nonparametric technique which has many applications. It is used to obtain estimates and standard errors for those estimates, as well as to estimate bias in Bootstrapping involves repeated estimates. resampling from the original sample of data. We have applied bootstrapping to the density estimation problem. Preliminary studies show that parallelizing the resampling portion of the bootstrap will yield significant improvements in processing time. Detailed results of our bootstrap study will be presented in a forthcoming paper.

5. Conclusions

We have shown that significant gains in efficiency may be had by parallelizing statistical computations. We have also presented a method for achieving fault detection in multiple regression parallelized computations. This method would be of some importance in maintaining a "stand-alone" system with automated input and processing which used multiple regression calculations in accomplishing its mission. We plan to extend the fault detection to other computations, such as kernel density estimation and bootstrapping.

Table 1
Results of Part 1 of the Timing Study
Effective Time

Observations per node per	Sample		Nu	mber of Ne	odes	
package	Size	1	_2	4	8	16
125	8000	15337	7722	3931	2058	1212
	16000	30703	15458	7854	4088	2266
250	8000	15337	7721	3935	2084	1301
	16000	30699	15448	7852	4106	2332
500	8000	15358	7733	3957	2155	1472
	16000	30737	15467	7877	4155	2496

Table 2
Results of Part 2 of the Timing Study

Observations	3		Effecti	ve Time	
per node per	Sample		Number	of Nodes	
_package	Size		1	16	Speed-up
125	8000		15337	1212	12.65
	16000		30703	2266	13.55
	32000		61471	4460	13.78
	64000		122132	8855	13.79
	128000		243648	17667	13.79
250	8000		15337	1301	11.79
	16000		30699	2332	13.16
	32000		61445	4521	13.59
	64000		122199	8905	13.72
	128000		243714	17685	13.78
500	8000	T	15358	1472	10.43
1	16000		30737	2496	12.31
	32000		61515	4661	13.20
	64000		122309	9043	13.53
	128000		244085	17815	13.70

REFERENCES

Draper, N. R. and H. Smith (1981), Applied Regression Analysis, New York: John Wiley and Sons.

Efron, B. (1982), The Bootstrap, the Jackknife and Other Resampling Plans, Philadelphia: Society for Industrial and Applied Mathematics.

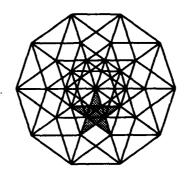
Francoise, A. (1985), Parallel Processing, Cambridge: MIT Press.

Meyer, Gerald G. L. and Howard L. Weinert (1984), "Parallel algorithms and computational structures for linear estimation problems," in *Statistical Signal Processing*, (E. Wegman and J. Smith, eds.), New York: Marcel-Dekker, Inc., 507-516.

Law, A. and K. David (1982), Simulation Modeling and Analysis, New York: McGraw-Hill.

Silverman, B. (1986), Density Estimation for Statistics and Data Analysis, London: Chapman and Hall.

Wegman, E.J. (1988), "Stochastic load balancing in parallel computers," Center for Computational Statistics Technical Report 30, George Mason University, Fairfax, VA.



The Fifth Distributed Memory Computing Conference

11: Full and Banded Matrix Algorithms

Solving Very Large Dense Systems of Linear Equations on the iPSC*/860

David S. Scott Enrique Castro-Leon Edward J. Kushner

Intel Scientific Computers 15201 NW Greenbrier Parkway Beaverton, Oregon 97006

Abstract

Certain engineering problems, such as radar cross section modeling, must solve systems of linear equations AX = B, where A is a large, dense, 128-bit complex matrix and B is the matrix of right hand sides. Solution to problems as large as 20000x20000 are needed now with even larger problems anticipated.

This paper describes an implementation of an out of core linear system solver on the Intel iPSC/860, a hypercube with Intel i860 based compute nodes and compatible Intel Concurrent I/O system. Block Gaussian elimination, with restricted pivoting, is used, paging blocks on and off the I/O system as needed. Gaussian elimination requires about 21.5 (64-bit real) Teraflops to solve the 20K problem, which would take 2.5 days on a 100 Mflop machine.

At both the cube and node levels, the basic operation, $C = C \cdot A*B$, has been carefully optimized. A hand coded i860 assembler kernel is used at the node level and asynchronous message passing and asynchronous disk I/O overlap almost all data movement with computation. A 64 node iPSC/860 with 6 I/O nodes and 12 disks factors a 20K problem in about four hours, sustaining 1.4 Gflops. Plans to extend the algorithm to even larger problems using tape storage will be described.

The iPSC/860 and the Concurrent File System

The iPSC/860, a distributed-memory messagepassing multicomputer, contains up to 128 separate compute nodes with optional I/O nodes and disks. Each compute node is an Intel i860 microprocessor with 8 Megabytes of memory and a FIFO-based interface to the Direct-Connect routing hardware. Each node injects and removes messages from the communication system at a rate of 2.8 Mbytes/sec.

Since the peak double precision performance of the i860 on matrix computations is 40 Mflops, a 64 node system has a peak performance of 2.56 Gflops. This assumes that all message passing and I/O required by the algorithm is completely overlapped with computation. The actual transfer of bytes between the FIFO interface and node memory is done by the i860 to maintain coherency of the on-chip cache. Thus, all message traffic involves some cycle stealing and peak performance will be unobtainable.

The optional I/O system on the iPSC/860 system provides parallel access to a set of SCSI disk drives controlled by Intel 80386 microprocessor-based I/O nodes. Messages between I/O nodes and compute nodes compete for the same wires which node-to-node messages use. Each I/O node can read or write a disk at about 1 Mbyte/sec. The Concurrent File System automatically spreads files across the available disks. All compute nodes independently open files, seek locations, and read data simultaneously. Data is transferred to and from the I/O system in separate 4K byte packets.

For a detailed description of the i860 see [1] and for a detailed description of the iPSC/860 see [2].

Block Gaussian Elimination

The Large Out-Of-Core Solver (LOOCS") code implements a variant of block Gaussian elimination. The matrix is divided into square submatrices called disk sections which are the units that are swapped off and on the disk. When A is a t x t matrix of disk

sections, the factor algorithm can be described as follows:

```
for i = 1, t
 for j = 1, i-1
                 //do ith row
 for k = 1, j-1
   Aij = Aij - Aik * Akj
  endfor
  Aij = Aij * Aii //Aii is already inverted
 endfor
 for j = 1, i-1
                 //do ith col
 for k = 1, j-1
   Aji = Aji - Ajk * Aki
 endfor
 endfor
 for j = 1, i-1
                 //do ith diagonal
  Aii = Aij - Aij * Aji
 Aii = inverse of Aii
endfor
```

The corresponding solve algorithm, for one block column of B, appears as:

```
for i = 1, t //forward elimination
for j = 1, i-1
Bi = Bi - Aij * Bj
endfor
endfor

for i = t, 1, -1 //back substitution
for j = t, i+1, -1
Bi = Bi - Aij * Bj
endfor
Bi = Aii * Bi
endfor
```

This variant makes all changes to a particular disk section at once. This helps minimize I/O since disk sections are written only once and means that the algorithm has natural checkpoints. Every section written to the disk is finished, except during the forward solve which can be handled by changing output files.

Only three types of operations are needed:

$$1. C = C - A*B$$

Explicit inversion of diagonal blocks is not necessary, but a matrix-matrix multiply can be parallelized much more efficiently than repeated forward elimination and backsubstitution, so even though the inversion is expensive in flops, it pays for itself in the later uses of the diagonal block.

This algorithm is potentially unstable since pivoting is done only inside diagonal disk sections. Unless the matrix is diagonally dominant, a diagonal disk section could be exactly singular causing the algorithm to fail. Ill conditioned diagonal disk sections are an indication of numerical instability. Unfortunately, the only guaranteed solution is to pivot down the whole column, which is much too expensive since most of the column is not in memory. However, the explicit inversion of the diagonal sections makes it easy to compute their condition numbers. This provides monitoring of the stability of the algorithm.

Square disk sections, as large as memory will allow, minimize I/O bandwidth requirements, since the work is proportional to the cube of the disk section size but the I/O is only proportional to the square.

Parallel Matrix Multiply

Both C = C - A*B and C = A*B are implemented simultaneously, by using the functionality of the BLAS3, see [3], routine ZGEMM which computes,

Equation (1).
$$C = alpha*A*B + beta*C$$
,

for specially chosen values of alpha and beta. The parallel matrix multiply routine is a folded version of a systolic algorithm for matrix-matrix product [4]. Assuming a $k \times k$ torus of processors, a subset of the hypercube topology, each disk section is divided into k^2 square node sections. At the beginning of a matrix multiply operation, each node in the mesh will have one specially selected node section of the A, B, and C disk sections. Each node will implement equation (1) on node sections while simultaneously passing its A section left and its B section up. If node sections are large enough, messages arrive before the node computation is finished so that the next node section multiply need not wait.

The disk section matrix multiply loop consists of k phases in which each node does the following:

Post receive from down
Post receive from right
Post send to left
Post send to right
Multiply
Wait for completion of messages

In the last (kth) phase no data is sent since there will be no further arithmetic.

To obtain the correct answer, it is important that the node sections of the A and B disk sections be carefully allocated to compute nodes. Node sections in the first row of A are assigned to their natural processors in the mesh. The next row is circularly shifted one position left. Each succeeding row shifts further left. Similarly, the first column of B is naturally assigned to the first column of processors. The second column shifts up one position. Each succeeding column of B shifts further up. Assuming a 4x4 processor mesh, Figure 1 shows the assignment to the mesh of the node sections of an A, B, and C disk section.

Parallel Matrix Inversion

Parallel matrix inversion is implemented using Gauss-Jordan inversion described in [5]. The computational step between communications is a

rank one update of the active submatrix in the form of equation (1), but with A and B as a single column and a single row respectively.

860 Optimization

To obtain optimal performance from the i860, a ZGEMM routine was hand coded in assembly language. The multiplication of the matrix A by a column of B is implemented as a sequence of complex ZAXPY operations, with the result accumulated in the data cache and written to the matrix C at the end. The code takes advantage of pipelined arithmetic and data reads, dual instruction mode, plus 128 bit reads and writes between registers and the data cache. The asymptotic speed of the kernel is 37.5 Mflops per node with an n-half of 10. For a more detailed description of this kernel see [6].

I/O Optimization

The I/O system must contain enough disks to store the matrix and I/O nodes to provide adequate transfer rate on and off those disks. When 64 i860 nodes solve a problem with 228x228 node sections, 6 I/O nodes are needed to provide adequate bandwidth. Ten disks would be enough to hold a 20K matrix, but since most systems have the same number of disks per I/O node, twelve were used in the benchmark system.

$$C_{11} = C_{11} \cdot A_{11} * B_{11} \qquad C_{12} = C_{12} \cdot A_{12} * B_{22} \qquad C_{13} = C_{13} \cdot A_{13} * B_{33} \qquad C_{14} = C_{14} \cdot A_{14} * B_{44}$$

$$C_{21} = C_{21} \cdot A_{22} * B_{21} \qquad C_{22} = C_{22} \cdot A_{23} * B_{32} \qquad C_{23} = C_{23} \cdot A_{24} * B_{43} \qquad C_{24} = C_{24} \cdot A_{21} * B_{14}$$

$$C_{31} = C_{31} \cdot A_{33} * B_{31} \qquad C_{32} = C_{32} \cdot A_{34} * B_{42} \qquad C_{33} = C_{33} \cdot A_{31} * B_{13} \qquad C_{34} = C_{34} \cdot A_{32} * B_{24}$$

$$C_{41} = C_{41} \cdot A_{44} * B_{41} \qquad C_{42} = C_{42} \cdot A_{41} * B_{12} \qquad C_{43} = C_{43} \cdot A_{42} * B_{23} \qquad C_{44} = C_{44} \cdot A_{43} * B_{34}$$

Figure 1. Initial Node Section Assignment For Matrix Multiply

The I/O performance was optimized in three ways. First, the required I/O was pipelined as much as possible, so that blocks needed during the next multiply were fetched during the previous one. Furthermore, writes of completed blocks were deferred until the last multiply for the next block when no fetch was performed. This allowed computation of the next block to begin without waiting for the previous block to be written.

Second, I/O nodes cache certain disk blocks in memory. When a compute node is reading a disk file, the I/O nodes try to preread disk blocks so that when a read request actually arrives, the desired block is already in memory. If too many nodes are reading at once, the memory of the I/O nodes is insufficient to keep the readahead blocks in memory until the actual read arrives. Such cache thrashing, which seriously degrades the I/O performance, was eliminated by increasing the size of the cache and limiting the number of nodes reading at any one time to 16. Similarly, writes were restricted so that only 8 nodes write at a time. This was accomplished by staging the reads and writes during different phases of the matrix multiply algorithm.

Finally, it was important to carefully locate the I/O nodes and select which sets of compute nodes are reading or writing at the same time. The iPSC/860 uses fixed routing to send messages to avoid deadlock. Dimensions in a 64 node cube are numbered from 0 to 5. Messages are routed in the lowest needed dimension first, working up to the highest needed dimension. Optimum I/O performance is hampered by contention for wires. Wire contention for messages headed to the same compute node is of no import since the messages are serialized at the node anyway. What must be avoided is wire contention for large messages which are headed for different nodes. Reads and writes are asymmetric. When reading, the large messages go from I/O nodes to compute nodes, so these paths need to avoid contention. When writing, the large messages go from compute nodes to I/O nodes so these paths are the critical ones.

The sixty four compute nodes can be arranged in an 8x8 mesh in which nodes in columns differ only in their three lowest bits and nodes in rows differ only in their three highest bits:

7	15	23	31	39	47	55	63
6	14	22	30	38	46	54	62
5	13	21	29	37	45	53	61
4	12	20	28	36	44	52	60
3	11	19	27	35	43	51	59
2	10	18	26	34	42	50	58
1	9	17	25	33	41	49	57
0	8	16	24	32	40	48	56

I/O nodes should be anchored to compute nodes in the same row, for example, the top row. Routing of messages from I/O nodes to compute nodes follows the hypercube routing down the column and then follows the hypercube routing across the row. Therefore two columns of nodes can read without any contention among paths from different I/O nodes to different compute nodes. The paired columns are (0,56), (8,48), (16,40) and (24,32).

When writing, the paths from the compute nodes to the I/O nodes are of interest. These paths go up columns followed by routing in the row of anchor nodes. Nothing can be done about contention in the row of anchor nodes, but one row of nodes should write at a time.

Performance

Performance of the parallel matrix multiply routine is summarized in Table 1. There is no communication on a single node. That column measures the performance of the assembly language routine. Degradation of performance for small problems on large machines is characteristic of message passing machines. The performance of 64 nodes for n=2048 is 35.9 Mflops/node which shows that little is lost to message passing overhead.

Table 1. Matrix Product Performance (Mflops)

Nodes				
Dimension	1	4	16	64
8	17.1	-	•	-
16	28.4	15	11	-
32	34.1	35	53	42
64	36.5	106	113	141
128	37.4	130	207	365
256	37.8	141	423	763
512	-	147	548	1379
1024	-	-	578	2165
2048	-	•	-	2300

The entire LOOCS code was timed on a 64 node iPSC/860 with 6 I/O nodes and 12 disks. Table 2 shows times and Megaflops obtained on four problems.

Table 2. Factorization Performance

Dimension	Seconds	Mflops
2500	146	285
5000	565	590
10000	2700	987
20000	15300	1394

Solve performance depends on the number of right hand sides. One right hand side is completely I/O bound. A full disk section of right hand sides can be solved at the same speed as the factorization.

Extensions to Tertiary Storage

Users wish to solve problems of order 100,000. The current algorithm requires 160 Gbytes of disk which is not cost effective. It is possible to extend the same hierarchical decomposition one more level to use a tertiary storage medium such as tape. Disk sections are aggregated into large square tape sections. The block algorithm is now applied to tape sections. The three types of operations among tape sections are implemented as a sequence of operations on disk sections. A total of 6 tape sections must fit on disk: current A, B, and C sections; one old C being written to tape; and two new A and B sections being fetched from tape. The tape sections should be made as large as possible subject to the constraint that six tape sections fit on the available disk space. Given the

transfer rate of 8mm video tapes, 12 I/O nodes with twelve disks and 6 I/O nodes with 12 8mm video tape drives provide sufficient I/O bandwidth from tape-to disk-to cube-to disk-to tape to keep a 128 node iPSC/860 compute bound. Such a machine could factor and solve a double precision complex system of linear equations of size 100,000 in about 8 days.

Conclusions

The LOOCS code running on the iPSC/860 obtains more than half the theoretical peak performance of the machine. It provides a fast and cost effective platform for solving large dense systems of linear equations.

References

- [1] Intel Corporation (1989), i860 64-bit Microprocessor Programmer's Reference Manual, Intel Corporation, Santa Clara, CA.
- [2] Lillevik, S. (1990), Touchstone Program Overview, This Proceedings.
- [3] Dongarra, J. J., Du Croz, J., Duff, I., & Hammerling, S. (1989), A Set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. on Math. Soft.
- [4] Duncan, K. (1990), A Survey of Parallel Computer Architectures, Computer, Vol 23, No. 2, p. 9.
- [5] Hipes, P. G. & Kupperman, A. (1988), "Gauss-Jordan Inversion with Pivoting on the Caltech Mark II Hypercube Multiprocessor", Proceedings of the Third Conference on Hupercube Multiprocessors, pp 1621-1634.
- [6] Scott, D. S. (1990), "A Fast i860 Matrix-Matrix Product Routine," Technical Report, Intel Scientific Computers, Beaverton, OR.

PARALLEL SOLUTION ALGORITHMS FOR THE TRIANGULAR SYLVESTER EQUATION

Apostolos Gerasoulis*
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903 USA

Izzy Nelken
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Abstract.

The parallelization problem can be divided into three main stages: identification of parallelism which includes dependency analysis, partitioning the statements into atomic tasks of granularity suitable to the target architecture and scheduling these tasks into the processors.

An MIMD coarse grained parallel algorithm is developed for the triangular Sylvester equation. We compare well known scheduling heuristics such as the naive and compute-ahead with the N-cp/misf methods which are described here. These methods trade off time and space according to the value of the parameter N. Our conclusion is that the N-cp/misf methods are faster than compute-ahead.

1 Introduction

1.1 Stages of parallelization

As mentioned in the abstract, the parallelization problem consists of three important stages:

- Identifying parallelism and finding the data dependencies.
- Partitioning the algorithm into indivisible tasks and and the data into corresponding data items.
 The size of the tasks depends on the problem as well as the target architecture.
- Scheduling the execution of these tasks and mapping the data items into a given multiprocessor.

In the first stage, the programmer or the compiler identifies parallelism at the finest possible grain. This parallelism is then represented by a data dependency graph. In the second stage, the data dependency graph is partitioned into tasks appropriate for the given granularity level of the target architecture. Under the convexity constraint, Sarkar [14], this results in a directed acyclic data dependency graph (DAG) in which all redundant edges have been deleted. For message passing architectures, the data must be distributed amongst the local memory of the processors. In this case the data must also be partitioned so that they are compatible with the task partitioning and the architecture. In the final stage, the data items are mapped and the execution of the partitioned graph is scheduled in the given multiprocessor. In this paper, we consider the problem of static list scheduling and data mapping for MIMD architectures such as hypercubes. For a more detailed description we refer the reader to Gerasoulis and Nelken [6] and Nelken [12].

1.2 The problem

Consider the matrix equation

$$AX + XB = C$$

where A, B and C are known $m \times m$, $n \times n$ and $m \times n$ real matrices respectively. The unknown X is also $m \times n$.

This equation is solvable if and only if A and -B have no eigenvalues in common, Golub et al. [7]. Henceforth, we will assume that the given matrix equation is solvable. A transformational solution method is based upon the equivalence of the original problem with

$$(U^{-1}AU)(U^{-1}XV) + (U^{-1}XV)(V^{-1}BV) = U^{-1}CV.$$

^{*}Supported by Grant No. 8706122 from NSF.

The transformational solution method consists of four stages, Golub et al. [7].

- 1. Transform A and B into a "simple" form by $A_1 = U^{-1}AU$ and $B_1 = V^{-1}BV$.
- 2. Compute $F = U^{-1}CV$.
- 3. Solve the transformed system $A_1Y + YB_1 = F$.
- 4. Compute $X = UYV^{-1}$, the solution to the original system.

In particular, the Bartles-Stewart algorithm, see [1], uses the transformations $A_1 = U^T A U$ and $B_1 = V^T B V$ where U and V are orthogonal matrices which are chosen so that A_1 and B_1 are upper quasi-triangular. A quasi-triangular matrix is triangular with possible 2×2 blocks along the diagonal.

In this paper, we will be concerned only with the third step of this procedure, solving the transformed system. We will assume that A_1 and B_1 are proper upper triangular matrices. Thus we are faced with the solution of a triangular Sylvester equation which is of the form AX + XB = C, where A and B are upper triangular matrices. For simplicity of presentation and analysis we also assume that m = n.

1.3 Parallel time

Our aim, of course, is to reduce the parallel time, T_p which is defined as the elapsed time of the processor which finishes last under the assumption that all processors begin at the same time.

During execution, the processor which finished last is either idle or working. The idle time is composed of:

- T_I Idle time due to synchronization of the data dependencies
- T_C Idle time due to communication of data
- T_D Idle time due to architectural constraints,
 e.g. bottlenecks and hot spots.

The working time is composed of:

• T_A - The arithmetic time

• To - The parallel program overhead.

Thus, the parallel time is given by the following sum

$$T_p = T_A + T_I + T_C + T_D + T_O.$$

In the shared memory case, T_C is substituted by T_L the memory latency time.

2 Identification of parallelism

The elements of X can be computed by elementwise identification:

$$x_{ij} = \frac{c_{ij} - \sum_{k=i+1}^{m} a_{ik} x_{kj} - \sum_{k=1}^{j-1} b_{kj} x_{ik}}{a_{ii} + b_{jj}}, \quad 1 \le i, j \le n.$$

If the equation is solvable, $a_{ii}+b_{jj}\neq 0$ and the division above can be performed. The x_{ij} 's must be found in a certain order as depicted in the structure in figure 1. $x_{n,1}$ is found first and is labeled by a 1. Then $x_{n,2}$ and $x_{n-1,1}$ can be computed in parallel, they are labeled by a 2. Afterwards, $x_{n-2,1}, x_{n-1,2}$ and $x_{n,3}$ can be found in parallel and they are labeled by a 3 and so on. Notice that all elements which are on the same diagonal can be computed in parallel.

$$\left(\begin{array}{ccccc}
4 & 5 & 6 & 7 \\
3 & 4 & 5 & 6 \\
2 & 3 & 4 & 5 \\
1 & 2 & 3 & 4
\end{array}\right)$$

Figure 1: A structure which shows the order in which the elements of X are solved.

An algorithm with less arithmetic operations results if we update the matrix C after computing each element of X. The resulting algorithm, called AXXBC, is:

- 1. Compute a matrix element $x_{ij} = \frac{c_{ij}}{a_{ii} + b_{jj}}$.
- 2. Update elements in the j'th column of C according to elements in the i'th column of A

$$c_{kj}=c_{kj}-a_{ki}\,x_{ij}, \quad 1\leq k\leq i-1.$$

3. Update elements in the i'th row of C according to elements in the j'th row of B

$$c_{ik} = c_{ik} - b_{jk} x_{ij}, \quad j+1 \le k \le n.$$

Let us define the following operations:

$$d(i,j): \quad x_{ij} = \frac{c_{ij}}{a_{ii} + b_{jj}}$$

$$u1(k,j): c_{kj} = c_{kj} - a_{ki} x_{ij}$$

and

$$u2(i,k): c_{ik} = c_{ik} - b_{jk} x_{ij}.$$

The fine grain data dependency graph is given in figure 2 for the case n=3. Because of our indexing scheme, the symbol u1(k,j) may appear several times, each time for a different value of the index *i*. Similarly, u2(i,k) may appear several times. The lines indicate data dependencies from top to bottom. For example, d(3,1) must be finished before any of u1(1,1), u1(2,1), u2(3,2) or u2(3,3) may begin execution. On the other hand, these four statements can be executed concurrently.

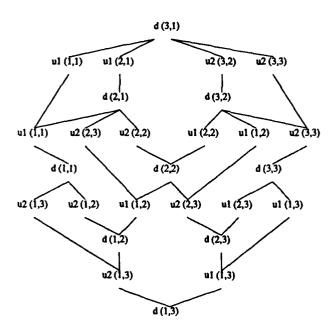


Figure 2: Fine grain data dependency graph, n = 3.

3 Partitioning

3.1 General

In this stage, we partition the fine grain operations into atomic or indivisible tasks whose granularity is suitable to the target architecture. Many partitioning are possible for the graph in figure 2. Sarkar [14] has imposed the "convexity constraint" on partitionings. A convex partitioning is one that satisfies the following conditions:

- A task can begin operating when all its inputs are available. It operates until completion and may produce outputs.
- Once a task is started it operates until completion without interruption.

The motivation is that non convex partitionings may lead to arbitrarily large communication and synchronization costs. Convex partitionings, on the other hand, have an acyclic coarse grain dependency graph that can be used on the macro-dataflow model.

3.2 According to rows

Many partitionings of the fine grain data dependency graph in figure 2 are possible. For example, algorithm SYLV_DIAG of Kägström et al. [10] uses a row oriented partitioning which is depicted in figure 3. Task (k, k) finds the k'th row of X and task (k, j) use the k'th row of X to modify the j'th row of C for j < k. In the figure, all statements which belong to a task are circumscribed and the task's name is written next to them. It is obvious that this partitioning is not suitable for the macro-dataflow model. For example, task (3, 2) may begin as soon as d(3, 1) has completed. However, under the rules of macro-dataflow, as described by Sarkar [14], it will have to wait until (3,3) has completed. Indeed, the SYLV_DIAG algorithm of [10], sends $x_{3,1}$ as soon as it has been computed and starts the execution of (3, 2).

The SYLV_DIST_B and SYLV_DIST_WB algorithms of [10] also use a similar row partitioning. However, in both these algorithms columns of X are mapped into the processors using the block and wrap mappings respectively. This means that the tasks of figure 3 are further divided. Each statement of an original task is to be executed in the processor which stores that element of X. The difference between the two partitionings stems from the different mappings. In SYLV_DIST_B, block mapping is used and $\frac{n}{p}$ contiguous elements of each row X are solved for by each

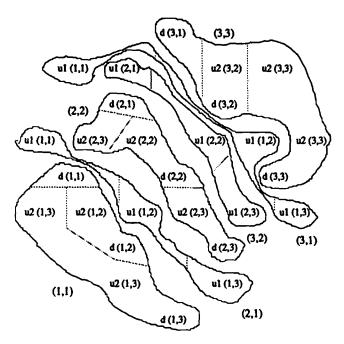


Figure 3: Row oriented partitioning which is used by SYLV-DIAG and a finer partitioning used by the other algorithms for n = 3 and p = 3 (data dependency lines have been removed for clarity).

processor. SYLV_DIST_WB, on the other hand, uses wrap mapping and sends each element of X as soon as it has been computed. Therefore, this algorithm uses messages of length 1. For a description of block and wrap mapping see Ortega [13].

In our example, n = 3 and p = 3, both block and wrap mappings are identical and so are the partitionings of SYLV_DIST_B and SYLV_DIST_WB. These are depicted in figure 3 by the dashed lines which divide each original task of SYLV_DIAG.

The SYLV_BLOCK algorithm divides the original matrix X into blocks and then solves for each block using another algorithm such as SYLV_DIAG. For our example, the blocks are of size 1×1 and the partitioning obtained is the same as that of SYLV_DIST_B and SYLV_DIST_WB.

3.3 According to diagonals

We consider the following diagonal partitioning which groups together all operations performed on the same diagonal. In figure 4 we show the partitioned fine grain data dependency graph. All tasks in a box marked

(k, j) belong to task T_k^j . For example, the task T_1^1 contains the statement d(3, 1) while the task T_2^5 contains the statements u1(1, 1), u2(2, 2), u1(2, 2) and u2(3, 3). Note the definition of (3, 3). We have lost the potential parallelism between d(1, 1), d(2, 2) and d(3, 3).

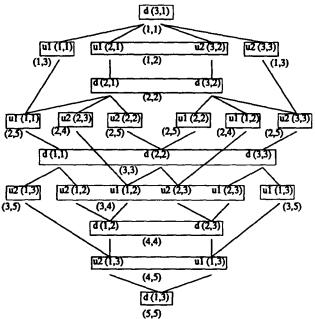


Figure 4: Partitioned data dependency graph. All tasks in a box marked (k, j) belong to task T_k^j .

The number of diagonals in a full $n \times n$ matrix is 2n-1. However, A and B are upper-triangular matrices and only have n diagonals numbered $n, n+1, \ldots 2n-1$. The number of elements in diagonal k is n-|n-k|. Task T_k^k in figure 4 consists of finding the k'th diagonal of matrix X which can overwrite the k'th diagonal of matrix C, it uses the n'th diagonal of matrices A and B. Task T_k^j uses diagonal k of X (which is stored and accessed as diagonal k of C) to modify diagonal j of C. It uses the (n+j-k)'th diagonals of both A and B.

Given the above partitioning, we find τ_k^j and τ_k^k the costs of executing tasks T_k^j and T_k^k respectively. These are measured in terms of the old Flops as defined by Golub and Van Loan [8]. The index arithmetic and housekeeping operations are not counted since we are only concerned with floating point operations.

$$\tau_k^k = n - |n - k|$$

$$\tau_k^j = \begin{cases} 2k & k < j < n \\ 2(n - (j - k)) & k < n \le j \\ 4n - 2j & n \le k < j. \end{cases}$$

After rearranging the nodes of the partitioned data dependency graph shown in figure 4 we obtain the DAG which is shown in figure 5 for n=4. In this figure, each task is represented by a circle. Inside the circle is its task id. To the right of the circle is the weight of the task, τ_k^j or τ_k^k . To the left of the circle is the level of the task which is defined in section 4.2.

A geometric comparison of this DAG with that of GE, see Gerasoulis and Nelken [6], reveals that the GE graph is wider in the beginning and then loses width one task at a time until, towards the bottom, both graphs become similar. Thus from a geometrical point of view, the partitioned GE has more potential parallelism than our DAG.

We can now compute a lower bound on the parallel time of any scheduling which uses this partitioning. Any scheduling must require at least the length of the longest path L(s). Also, using p processors, we can not expect to execute faster than $\frac{T_1}{p}$ where T_1 is the sequential time of the algorithm. Thus an obvious lower bound is:

$$T_p^{bound} = \max\{L(s), \frac{T_1}{p}\} = \max\{3n^2 - 2n, \frac{n^3}{p}\}.$$

3.4 Summary

The diagonal oriented partitioning conforms to the macro-dataflow model. However, the row oriented partitioning is used in a way which violates it. Sarkar [14] asks whether or not it is better to adhere to the macro-dataflow model. He mentions that specific experiments will have to be conducted to answer this question. The results of this paper can be seen as a step in this approach.

4 Scheduling

4.1 General

Under the assumption of zero communication cost we examine the CP/MISF scheduling of Kasahara and Narita [11] to the partitioned DAG of figure 5. Then

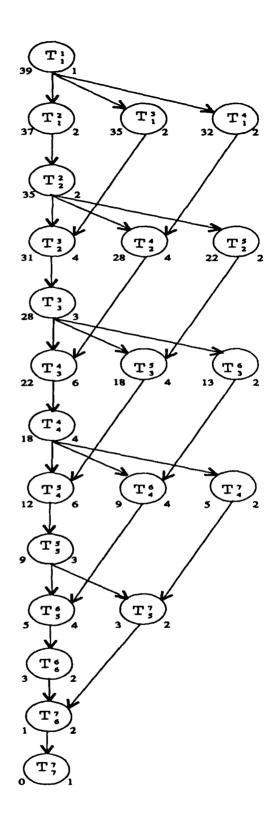


Figure 5: The DAG for n = 4.

we assume non-zero communication costs and briefly describe a four step scheduling methodology.

4.2 CP/MISF

Description

Kasahara and Narita's [11] CP/MISF scheduling is one of the best scheduling heuristics of a general DAG when communication costs are assumed to be zero. This heuristic has three stages:

- 1. Determine the level for each node. The level of a node is the longest path length from the node to the terminal node and a path length is the sum of all the task weights in the path. In figure 5 the weights and levels have already been determined.
- 2. Construct a priority list of the tasks. Under the CP/MISF rules, the tasks are sorted in descending order of levels. If two nodes have the same level then the task with most immediate successors has a higher priority. Ties are broken according to lexicographic ordering. Thus we sort the tasks based of the triad [level | number of successors | lexicographic order].
- 3. Perform list scheduling on the priority list. Whenever any processor becomes available, it scans the priority list from left to right and picks up the first task which is ready to be executed (i.e. all its predecessors have completed). A task which has been picked up for execution is marked as taken to avoid picking it up again.

Performance

To measure the performance of a scheduling we define R, the ratio of goodness

$$R = \frac{T_p^{bound}}{T_p}.$$

In figure 6, we plot the ratio of goodness, R, for the CP/MISF method assuming that communication costs are zero.

The performance of the method is indeed remarkable. It is within 1% of the lower bound, which leads us to ask the following question:

Is CP/MISF an asymptotically optimal method for the above problem?

It is obvious that for a realistic message passing architecture with non-zero communication costs, CP/MISF

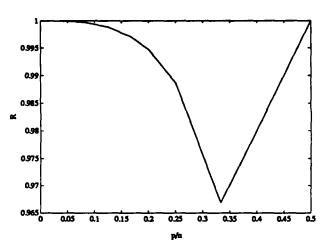


Figure 6: The ratio R for CP/MISF assuming $T_C = 0$ for n = 240.

will perform poorly because of its unacceptably high communication requirements. Even for shared memory architectures, say with a bus and local memory, its performance could deteriorate because of high data movement.

5 A scheduling methodology

The scheduling problem for message passing architectures which include communication cost is very difficult. We propose the following four step heuristic approach:

1. Clustering:

Find a "good" schedule for an unbounded number of virtual processors connected as clique. Because of the existence of communication cost, this stage will generate clusters of tasks that must be executed by the same processor on the target architecture, Sarkar [14]. The locality assumption, Gerasoulis and Nelken [5], which specifies that a processor may modify only the data which is stored in that processor, automatically determines the clustering. For our example, we obtain the following clusters:

$$M_1 = \{T_1^1\}, M_2 = \{T_1^2, T_2^2\}, \ldots$$

$$M_{2n-1} = \{T_n^{2n-1}, T_{n+1}^{2n-1}, \dots, T_{2n-1}^{2n-1}\}.$$

2. Physical mapping:

The 2n-1 clusters must be mapped to the p physical processors. Each processor will be assigned the tasks of several clusters in an attempt to load balance

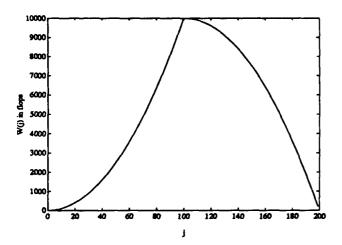


Figure 7: Work profile for n = 100.

the arithmetic work. With each cluster M(j), we associate a work load W(j) where

 $W(j) = \text{arithmetic work in } M_j$.

The work-profile of the clusters, George et al. [4], is a graph of W(j) against j, see figure 7.

It can be seen from the work profile that we can completely load balance the arithmetic provided that we use the wrap mapping which completely load balances the arithmetic if 2n-1 is a multiple of p. The wrap mapping also conforms to the proximity assumption, see Nelken [12], which further reduces communication costs.

3. Storage of data:

To reduce communication further the data items that are accessed most by the tasks in each processor are stored in that processor. All data items must be stored in the processors before execution begins. Our algorithm has four matrices to be considered: A, B, C and X. Since matrix X overwrites C, we will only consider the three matrices A, B and C.

We will postpone dealing with the storage of A and B until the next section on ordering. As for the matrix C, we associate data items (i.e. diagonals) of matrix C with the clusters. Each cluster M_j is associated with the data item which it accesses most which is the j'th diagonal of C. The locality assumption implies the definition of M(j) and that data item j should be stored in the same processor which executes cluster M(j). The data mapping of matrix C is obvious. Since we have used wrap mapping of the

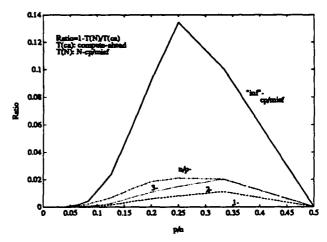


Figure 8: N-cp/misf vs. compute-ahead for AXXBC with wrap mapping and n = 240.

diagonals of C to the processors.

4. Task ordering:

In this stage, the tasks assigned to each processor are ordered and execution threads are formed. In a static scheduler, these threads are determined at compile time. In the next section, we will describe the N-cp/misf ordering methods and compare them with compute-ahead.

6 The N-cp/misf methods

In this section we define the N-cp/misf methods. We impose the memory constraint and assume that a_i diagonals are mapped to each processor p_i and that the same processor has enough space to store an additional b_i diagonals. For simplicity, assume that $a_i = (2n-1)/p$ and $b_i = N$ for $0 \le i \le p-1$. Thus N is the number of additional data items that can be stored in each processor.

Our approach is to form p priority lists by sorting the tasks assigned to each processor. The tasks assigned to each processor are sorted according to the CP/MISF criteria, see section 4.2. We then use a modified list scheduler (MLS) whose input are the p priority lists and the parameter N, and whose output is a scheduling which satisfies the DAG dependencies, locality assumption and memory constraint.

The N-cp/misf methods are derived as follows:

 Computation of levels: Find the levels for the DAG as in the traditional CP/MISF method.

- 2. Sorting: The p groups of tasks mapped in each processor are sorted according to the CP/MISF criteria. Tasks with higher levels are placed in front of tasks with lower levels. If several tasks have the same level, then they are sorted according to the number of outgoing edges. If several tasks have the same level and the same number of outgoing edges, they are sorted lexicographically. Thus our sort is again based on the triad [level | number of successors | lexicographic order] in each processor. These are the p priority lists.
- 3. Modified list scheduler: In the MLS each available processor scans its priority list from left to right and executes the first task that satisfies the following conditions:
 - The task is ready to be executed.
 - The execution of the task will not result in a scheduling that requires space for more than N data items.

The first task in the priority list which satisfies both constraints is scheduled to the processor. If no such task exists, the processor remains idle.

The MLS could deadlock if the execution of a task is ready but requires N+1 space and a descendant that could break the deadlock depends on this task. The deadlock can be broken by re-receiving the same data. We have not observed this situation in our DAGs and conjecture that it does not occur.

Note that there is a range of N-cp/misf methods. For the 1-cp/misf method each processor needs at the most space for 1 additional data item to execute. For $N = \frac{n}{p}$ the space requirements for each processor are doubled. At the extreme is the "inf"-cp/misf method which assumes that each processor has infinite memory.

In figure 8, we compare the N-cp/misf methods with the compute-ahead rdering in terms of $T_A + T_I$ for both approaches. We plot the quantity 1 - T(N)/T(ca) where T(N) is the $T_A + T_I$ time for the N-cp/misf method and T(ca) is the corresponding time for compute-ahead. The graph represents the savings achieved by

the N-cp/misf method. 1-cp/misf has the same performance as compute-ahead but "inf"-cp/misf is up to 14% faster.

7 Comparing with SYLV_DIAG

Kägström's algorithms have been implemented on Dunigan's [2] message passing multiprocessor simulator (PP-SIM). We have also implemented the naive and compute-ahead algorithms on PPSIM. All implementations are in single precision floating point C. The experimental results are for the case m=n=64. Kägström's implementations as well as our own, use initial simulator values which correspond to the Intel iPSC cube.

cube_init(0.1,0.3,0.2,1024);

The speedup of compute-ahead has been plotted in figure 9. It should be compared with figures 4.a of [10]. The speedup of SYLV_DIAG has been reproduced in figure 9 from data given to us by Kägström [9] for comparison purposes. There are two versions of the compute-ahead program:

- The first version, compute-ahead with local copies
 of A and B, is known as the store approach,
 copies of all the diagonals of A and B are stored
 in each processor.
- The second version, compute-ahead with communication of A and B, is known as the mix approach, only copies of the main diagonals of A and B are stored in each processor while all other diagonals are communicated as required.

Note the time space trade-off between the two versions of the compute-ahead program. The store version exhibits a better speedup but has higher local memory requirements. The mix version of the program exhibits a poor speedup due to the fact that diagonals of A and B are also communicated rather than just diagonals of C.

It should be noted that in Dunigan's PPSIM simulator [2] the communication delay of a message of length M sent across h hops is

sM + hrM

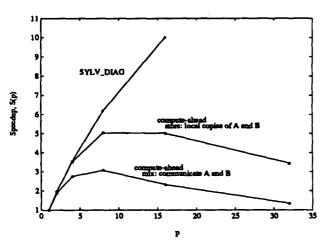


Figure 9: The speedups of compute-ahead and SYLV_DIAG, m = n = 64.

where s is the startup delay value, r is the communication delay for a floating point number and M is rounded up to the nearest packet size. In the compute-ahead algorithm, h is always 1 since only nearest neighbor communication is used. In this case, the PPSIM delay is (s+r)M.

On real hypercubes, such as the NCUBE, the delay for a message of size M communicated between neighbors is given by $\alpha + \beta M$, see Dunigan [3], an additional startup factor of α . Usually $\alpha \gg \beta$ and therefore it may be that on a real machine, the penalty for sending short messages, as is done in Kagström's algorithms, would be worse than it is in the simulator.

It should be pointed out that the results in figure 9 are for a small case, i.e. n = 64. If n and p are large the results may be different since the diagonal program requires overhead which is of low order. Subroutine T_k^j , for example, has index arithmetic which is done once each time it is called. If n is large, the effect of this overhead will be less noticeable.

8 Further research

Consider the N-cp/misf orderings. Unless we use the store version of the program, we would have to communicate the diagonals of A and B. The communication strategy would have to be modified for these orderings. For example, in order to determine which diagonals of A and B will be needed by each processor when using the "inf"-cp/misf ordering method, one needs to know the scheduling explicitly. Further,

the communication strategy used for the naive and compute-ahead orderings, where each processor sends all of its diagonals to its neighbor, might have to be changed.

The low speedups of the diagonal algorithms are due in part to the fact that the partitioned graph, which appears in figure 5, has less potential parallelism than that of the non-convex partitioning used by SYLV_DIAG. This is a partial response to Sarkar's question [14] about the restrictiveness of the macrodataflow model. In this case, the convex partitioning does not exhibit enough parallelism and it might be better to use a non-convex one. More accurate analysis and actual machine tests are needed to determine which of the alternative approaches is better.

References

- [1] R.H. Bartles and G.W. Stewart. A Solution of the Equation ax + xb = c. Communications of the ACM, 15:820-826, 1972.
- [2] T.H. Dunigan. A Message Passing Multiprocessor Simulator. Technical Report ORNL/TM-9966, Mathematical Sciences Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, May 1986.
- [3] T.H. Dunigan. Hypercube Performance. In Hypercube Multiprocessors, pages 178-192. SIAM, 1987.
- [4] A. George, M.T. Heath, and J. Liu. Parallel Cholesky Factorization on a Shared Memory Processor. Lin. Algebra Appl., 77:165-187, 1986.
- [5] A. Gerasoulis and I. Nelken. Gaussian Elimination and Gauss-Jordan on MIMD Type Architectures. Technical Report LCSR-TR-105, Department of Computer Science, Rutgers University, New-Brunswick, NJ 08903, May 1988.
- [6] A. Gerasoulis and I. Nelken. Scheduling Linear Algebra Parallel Algorithms on MIMD Architectures. Technical Report LCSR-TR-122, Department of Computer Science, Rutgers University, New-Brunswick, NJ 08903, May 1989.

- [7] G.H. Golub, S. Nash, and C.F. Van Loan. A Hessenberg-Schur Method for the Problem AX + XB = C. IEEE Trans. on Automatic Control, AC-24(6), December 1979.
- [8] G.H. Golub and C.F. Van Loan. Matrix Computations Second Edition. Johns Hopkins, 1989.
- [9] B. Kägström. Private communication.
- [10] B. Kägström, L. Nyström, and P. Poromaa. Parallel Algorithms for Solving the Triangular Sylvester Equation on a Hypercube Multiprocessor. Technical Report UMINF-136.87, ISSN-0348-0542, Institute of Information Processing, University of Umeå, Umeå, Sweden, December 1987.
- [11] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Trans. on Computers*, C-33:1023-1029, 1984.
- [12] I. Nelken. Parallelization for MIMD Multiprocessors with Applications to Linear Algebra Algorithms. PhD thesis, Laboratory of Computer Science Research, Rutgers University, New Brunswick, NJ 08903, October 1989.
- [13] J.M. Ortega. Introduction to Parallel and Vector Solution of Linear Systems. Plenum, New York, 1988.
- [14] V. Sarkar. Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. PhD thesis, Department of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305, 1987. Also a book in the series Research Monographs in Parallel and Distributed Computing, The MIT Press, 1989, Cambridge MA.

Reducing Inner Product Computation in the Parallel One-Sided Jacobi Algorithm

Charles Romine*
Mathematical Sciences Section
Oak Ridge National Laboratory

Kermit Sigmon[†]
Department of Mathematics
University of Florida

Abstract

One drawback of the cyclic one-sided Jacobi algorithm is the necessity of computing the inner product of each column pair to be rotated—indeed, just to check whether the pair needs to be rotated. However, if the orthogonality of a column pair known to be orthogonal during a sweep is not subsequently destroyed before being encountered during the next sweep, then the inner product computation in the latter sweep is unnecessary. The number of such inner products becomes significant as the process nears convergence. To avoid these unnecessary inner products, the usual algorithm is extended to include a data structure which keeps a record of the current orthogonality status of the column pairs.

In the parallel setting, the reduction during the latter sweeps in both the number of column pairs rotated and the number of inner product computations in a sweep will not generally be uniformly distributed across the processors. Because of the resulting loss of load balance, the actual improvement in runtimes will be less than one would otherwise expect. A statistical analysis of this phenomenon is given.

The performance of the enhanced algorithm is observed through implementation on the 128-node Intel iPSC/860 at the Oak Ridge National Laboratory. A discussion of the correlation of these results with the statistical analysis mentioned above is also given.

1 Introduction

It was observed some time ago by Sameh [12] and Luk [7] that the one-sided Jacobi algorithm was well-suited for singular value and eigenvalue computation in a multiprocessor environment. More recently it has been observed [1,4,5,6,10] that the one-sided Jacobi algorithm is especially naturally suited for such computations on distributed memory and vector architectures. In the distributed memory environment, its natural parallelism permits excellent load balancing and the required message passing is relatively small. Because of its richness in vector operations, it is naturally suited for vector architectures.

This has led to a renewed interest in the one-sided Jacobi algorithm. Berry and Sameh [1] have effectively implemented it in a multiprocessor environment. Eberlein [4,5] and Eberlein and Park [6,8] have done so on distributed memory machines. Rath [9] and de Rijk [10] have given a fast Givens-type variant of the one-sided Jacobi algorithm which reduces the operation count and permits more effective vectorization.

The Jacobi algorithms have other advantages. It has recently been shown by Demmel and Veselić [3] that the Jacobi algorithms compute small eigenvalues and singular values and small components of eigenvectors and singular vectors with higher relative accuracy than either the QR and Golub-Reinsch algorithms or the divide and conquer method.

One drawback of the cyclic one-sided Jacobi algorithm is the necessity of computing the inner product of each column pair to be rotated—indeed, just to check whether the pair needs to be rotated. However, if the orthogonality of a column pair known to be orthogonal during a sweep is not subsequently destroyed before being encountered during the next sweep, then the inner product computation in the latter sweep is unnecessary. The number of such inner

^{*}The work of this author was supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-840R21400 with Martin Marietta Energy Systems Inc.

[†]The work of this author was supported in part by Oak Ridge National Laboratory, contracts 32X-SA783V and 95X-SC556V, and by Oak Ridge Associated Universities, DOE contract DE-AC05-76OR00033.

products becomes significant as the process nears convergence.

We extend the usual algorithm so that most of these unnecessary inner products are avoided. This is done by use of a data structure which keeps a record of the current orthogonality status of the column pairs.

During the latter sweeps of the one-sided Jacobi process, there is a reduction in both the number of column pairs which must be rotated and, as noted above, the number of necessary inner products. In the parallel setting, these reductions will not generally be uniformly distributed across the processors. Because of the resulting loss of load balance, the actual improvement in runtimes will be less than one would otherwise expect. A statistical analysis of this phenomenon is given.

The performance of the enhanced algorithm is observed through implementation on the 128-node Intel iPSC/860 at the Oak Ridge National Laboratory. An analysis of the correlation of these results with the statistical analysis mentioned above is also given. The results indicate that one can expect about a 10-20% improvement in performance for a small number of processors, with this improvement degrading as the number of processors increases due to the statistical loss of load balance mentioned above.

2 The One-Sided Jacobi Algorithm

The singular value decomposition of a real $m \times n$ matrix $A, m \ge n$, may be given by

$$A = U\Sigma V^T$$

where

$$U^T U = I_n = V^T V$$
 and $\Sigma = diag(\sigma_1, \dots, \sigma_n)$

with $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_n \geq 0$. The diagonal entries σ_i of Σ are called the singular values of A. The columns of U and V are called the left and right singular vectors (respectively) associated with the respective singular values of A. Since $AA^T = U\Sigma\Sigma^TU^T$ and $A^TA = V\Sigma^T\Sigma V^T$, these singular vectors are orthonormalize i eigenvectors associated with eigenvalues of AA^T and A^TA , respectively, and the singular values are the square roots of the eigenvalues of A^TA .

The one-sided Jacobi Algorithm for computing the singular values proceeds as follows.

Given $A \in \mathbb{R}^{m \times n}$ with $m \ge n$, one first computes an orthogonal matrix V such that the columns of $\tilde{U} := AV$ are orthogonal. If one then sets $\Sigma :=$

 $diag(\sigma_1, \ldots, \sigma_n)$, where σ_i is the 2-norm of the *i*-th column of \tilde{U} , and scales the columns of \tilde{U} by the σ_i to form U, then $\tilde{U} = U\Sigma$ and hence

$$A = U \Sigma V^T$$
 and $U^T U = I_n$.

Thus, the SVD is obtained if such a matrix V can be computed.

The matrix V can be computed iteratively as a product of planar rotation matrices, each of which rotates a pair of columns of A, as follows.

Given an ordered pair $[a_i, a_j]$ of columns from A, a plane rotation can be determined such that if

$$[a_i^{\bullet}, a_j^{\bullet}] := [a_i, a_j] \begin{bmatrix} c & -s \\ s & c \end{bmatrix},$$

then a_i^* and a_j^* are orthogonal and $||a_i^*|| \ge ||a_j^*||$. (Here and in the sequel, $||\cdot||$ denotes the two-norm). An algorithm for rotating a pair of columns a_i and a_j of A is the following. One must first select a tolerance ϵ to use in the test for orthogonality.

Algorithm Rotate

- A. If $\frac{a_i^T a_j}{\|a_i\| \|a_j\|} < \epsilon$, then a_i and a_j are taken to be already orthogonal so set $a_i^* := a_i$ and $a_j^* := a_j$ provided $\|a_i\| \ge \|a_j\|$ set $a_i^* := a_j$ and $a_j^* := a_i$ otherwise.
- **B.** If $\frac{a_i^T a_j}{\|a_i\| \|a_j\|} \ge \epsilon$, then
 - 1. Compute c and t:
 Compute $t := |\tau| + \sqrt{1 + \tau^2}$,
 where $\tau := \frac{\|a_1\|^2 \|a_1\|^2}{2a_1 \tau a_1}$.

 If $\|a_i\| \ge \|a_j\|$, set $t := \frac{1}{t}$ if $\tau \ge 0$ and $t := -\frac{1}{t}$ if $\tau < 0$;
 otherwise, set t := -t if $\tau \ge 0$.
 Compute $c := \frac{1}{\sqrt{1+t^2}}$.
 - 2. Rotate a_i and a_j : $a_i^* := c(a_i + ta_j)$ $a_j^* := c(a_j - ta_i)$.
 - 3. Update $||a_i||^2$ and $||a_j||^2$: $||a_i^*||^2 := ||a_i||^2 + ta_i^T a_j$ $||a_j^*||^2 := ||a_j||^2 - ta_i^T a_j$.

The one-sided Jacobi algorithm is performed in sweeps, each of which consists of rotations of each of the N := n(n-1)/2 possible pairs of columns performed in some fixed sequence with the order of each pair $[a_i, a_j]$ chosen to ensure a specified order

Note that for $x, y \in \mathbb{R}^n$, if u = c(x + ty) and v = c(y - tx), then $u^T u = x^T x + tx^T y + tu^T v$ and $v^T v = y^T y - tx^T y - tu^T v$, where $c = \cos \theta$ and $t = \tan \theta$ are chosen as in Algorithm Rotate.

 $||a_i^*|| \ge ||a_j^*||$ of the result. The Jacobi algorithm is characterized by this order, called a *Jacobi ordering*.

The sweeps are repeated until the columns are pairwise orthogonal as measured by ϵ in Algorithm Rotate. Pairwise orthogonality is usually determined by counting, for each sweep, the number of column pairs that were already orthogonal when encountered; when that number is N, convergence is declared (actually, N-1 will suffice; see §5).

To rotate a column pair $[a_i, a_j]$ one needs to have available $||a_i||$, $||a_j||$, and the inner product $a_i^T a_j$. However, one need not recompute the column norms for each new pair. After initializing an n-vector to contain the squares of the norms of the columns, this vector can be easily updated after each rotation as in step B3 of Algorithm Rotate. When convergence is achieved, this vector will contain the squares of the singular values of A.

Unlike the column norms, the inner product $a_i^T a_j$ must be computed for each rotation. These inner products form a significant portion of the floating point operations in the one-sided Jacobi algorithm. It is the reduction of these inner product computations that we address in §4 and §5.

3 The Parallel Algorithm

Rotations of pairs of columns with disjoint index pairs are independent. Therefore, if index pairs in a Jacobi ordering are disjoint, the corresponding rotations can be performed in parallel.

In the parallel Jacobi algorithm the rotations of a sweep are partitioned into groups of independent rotations with each group of rotations performed in parallel. There exist many parallel Jacobi orderings in which the maximum number $\lfloor n/2 \rfloor$ of rotations is performed in each of the n-1 (n, if n is odd) time steps. It is such optimal orderings that we use in our algorithm.

Our interest is in implementation of the parallel algorithm on a ring-connected distributed-memory multiprocessor. The implementation we use follows that introduced by Eberlein in [5] and futher discussed in Eberlein and Park [6].

The columns of the matrix are partitioned into 2p blocks of size as uniform as possible, where p is the number of processors. The algorithm assumes that $2p \le n$. The blocks are distributed to the processors in pairs as indicated in Figure 1 for p=4. Each column is extended by one entry, which will contain the square of the two-norm of the column. After initialization, this entry is updated with each rotation as in Algorithm Rotate; upon convergence, these entries

will contain the squares of the singular values.

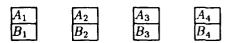


Figure 1: Distribution of columns (p = 4)

One begins each sweep by rotating all pairs of columns within each block in a lexiographically cyclic order. Then, after rotating each column of block A with each column of block B in each processor, the blocks are redistributed among the processors according to the communication pattern indicated in Figure 2 for p=4. The columns of block A are again rotated with the columns of block B in each processor. The sweep is completed after 2p-1 such "subsweeps."

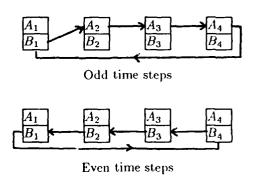


Figure 2: Communication pattern

Note that under this communication pattern, all information for computing the rotation parameters is available in the processor containing the column pair so the columns can be rotated locally. Hence, the amount of communication is small, with only one send required in each subsweep, or 2p-1 per sweep. Furthermore, the load balance remains exceptionally uniform.

4 Pairwise Decoupling

For each column pair encountered during a sweep, one first computes the inner product of the columns and then, if they are not already orthogonal, rotates the columns. For an $m \times n$ matrix, each inner product costs m multiplications and the rotation about 4m multiplications.² In the latter sweeps, the inner product computation becomes dominant since many

²The count for a rotation is 2m if fast rotations are used (see [9,10])

of the column pairs encountered are already orthogonal. A typical count of the number of orthogonal pairs encountered in a sweep is illustrated by "Orthogonal" in Table 1 for a random 128×128 matrix with p=4.

Table 1: n = m = 128, p = 4

Sweep Orthogonal Decoupled 1 0 0
1 0
1 0
2 0 0
3 0 0
4 0 0
$0 \qquad 0$
6 1 0
7 1313 0
8 7252 98
9 8074 3128
10 8128 8069

If the orthogonality of a column pair known to be orthogonal during a given sweep is not destroyed before being encountered during the next sweep, then the inner product computation in the latter sweep is unnecessary. To avoid these redundant inner products, we extend the algorithm to include a data structure which keeps a record of the current orthogonality status of the column pairs. In the example given in Table 1, the column "Decoupled" gives the number of such inner products which could be avoided in this manner.

To extend the algorithm, each column is assigned a column number (this is unecessary in the original algorithm) and extended to include an n-vector of flags. The vector of flags of a column remains with the column as it is sent to the various processors. Orthogonality of column i with column j is indicated by both the j-th flag of column i and the i-th flag of column j being set. Requiring both flags to be set permits one to avoid additional interprocessor communication.

The flag management is as indicated in the following extension of Algorithm Rotate.

Algorithm Rotate (with pairwise decoupling)

- A. If flags indicate the columns are orthogonal, do nothing. Otherwise,
- **B.** If $\frac{a_1^T a_1}{\|a_1\| \|a_1\|} < \epsilon$, then
 - 1. Interchange columns i and j if $||a_i|| < ||a_i||$.
 - 2. Set the j-th flag of column i and the i-th flag of column j.
- C. If $\frac{a_1^T a_1}{\|a_1\|\|a_2\|} \ge \epsilon$, then

- 1. Rotate columns i and j.
- 2. Set the j-th flag of column i and the i-th flag of column j.
- 3. Clear all the other flags of columns i and j.

Of course, there is some overhead associated with maintaining this data structure. Thus, its implementation should be delayed until that sweep where the savings in inner product computation exceeds the cost of this overhead. Our experience indicates that beginning implementation with the sweep following that in which "Orthogonal" becomes positive is appropriate. Clearly there is nothing to be gained by starting its implementation before this point.

For reasons given in §7, the actual improvement in runtimes due to the reduction during the latter sweeps of either the number of column pairs rotated or the number of inner product computations is likely to be less that the figures in Table 1 might suggest.

5 Subspace Decoupling

The reduction of inner products described in the preceding section can be further improved. Suppose H and K are disjoint collections of columns for which each column in H is orthogonal to each column in K. If two columns in H are rotated, then the orthogonality of the resulting columns with the columns in K is preserved since K is contained in the orthogonal complement of the subspace generated by H, and the rotated columns, which are linear combinations of the original columns, remain in this subspace.

In the preceding section it was assumed that when two columns are rotated, the orthogonality of the resulting columns with any other column could no longer be assured. However, in view of the subspace decoupling noted above, if a column is orthogonal to each of a pair of columns to be rotated, then its orthogonality with the updated pair is assured after rotation.

The data structure of the preceding section can be revised to reflect this preservation of orthogonality. The following further revision of Algorithm Rotate indicates how the flags can be managed to implement this feature.

Algorithm Rotate (with subspace decoupling)

- A. If flags indicate the columns are orthogonal, do nothing. Otherwise,
- **B.** If $\frac{a_1^T a_1}{\|a_1\| \|a_2\|} < \epsilon$, then
 - 1. Interchange columns i and j if $||a_i|| < ||a_j||$.

2. Set the j-th flag of column i and the i-th flag of column j.

C. If
$$\frac{a_i^T a_j}{\|a_i\| \|a_j\|} \geq \epsilon$$
, then

- 1. Rotate columns i and j.
- 2. Set the j-th flag of column i and the i-th flag of column j.
- 3. For each k other than i and j,
 - a) If either of the k-th flags of of columns
 i and j are clear, clear both of them.
 - b) If both of the k-th flags of columns i and j are set,
 - if both the i-th and j-th flags of column k are set, do nothing.
 - (ii) Otherwise, clear both the i-th and j-th flags of column k.

Implementation of this enhancement in a distributed memory environment appears to be problematic; it requires either significant additional interprocessor communication or, within the existing communication pattern, a very large message size. It may be more suitable for other environments. The authors plan to explore the feasibility of the use of subspace decoupling in a subsequent paper.

One interesting consequence of subspace decoupling is that when N-1 (rather than N) of the column pairs encountered during a sweep are already orthogonal, then convergence can be declared. For if all except one pair is orthogonal, when that pair is rotated to become orthogonal, its orthogonality with all other columns is preserved because of subspace decoupling. Hence, upon completion of the sweep, all N pairs are orthogonal.

6 Ordering the Singular Values

As noted in §2, when one rotates the column pair $[a_i, a_j]$ to produce the orthogonal column pair $[a_i^*, a_j^*]$, the angle of rotation can be chosen to ensure that $||a_i^*|| \ge ||a_j^*||$. Hence, for some fixed ordering of the columns, the rotations can always be chosen in the one-sided Jacobi algorithm so that $||a_i^*|| \ge ||a_j^*||$ whenever i < j. It has been suggested that the rotation angles be chosen in this manner to cause the computed singular values to be ordered.

This choice of rotation angles does, in fact, cause the singular values, upon convergence, to be ordered. Since we are aware of no proof of this result in the literature, we supply a proof in [11]. We note, however, that showing that the computed singular values are ordered is complicated by the fact that, prior to convergence, the column norms need not be ordered. In particular, when multiple—or nearly equal—singular values are present, this complication is evident. As an illustration, consider the matrix

$$A = \begin{bmatrix} \sqrt{3} & 1 & -2\\ \sqrt{3} & -2 & 1\\ \sqrt{3} & 1 & 1 \end{bmatrix}$$

If a sweep on A is performed by rotating the column pairs in the order (1,2),(1,3),(2,3), then before the sweep the column norms are $3,\sqrt{6},\sqrt{6}$ and afterwards they are $3,\sqrt{11},1$.

We now turn to a description of how to ensure the appropriate choice of rotation angles in the parallel algorithm with the communication pattern given in §3. This choice of rotation angles is incorporated into our algorithm.

We assume a fixed order of the columns of the matrix in terms of the initial distribution of blocks to the processors as follows: Within blocks, the columns have their natural order; between blocks, the columns have the order given by the ordering

$$A_1, B_1, A_2, B_2, A_3, B_3, \ldots, A_p, B_p$$

of the blocks, where the blocks are initially distributed to the processors as indicated in the top half of Figure 3.

We first note that under the communication pattern of $\S 3$, after their initial distribution to the processors, the blocks of columns do not return to their original location until after two sweeps. The location of the blocks after each sweep is illustrated in Figure 3 for p=4.



After even-numbered sweeps

$$egin{array}{c|cccc} \hline A_1 & & & \hline B_4 & & & \hline B_3 & & & \hline B_2 \\ \hline B_1 & & & A_4 & & & A_3 & & \hline A_2 & & & \end{array}$$

After odd-numbered sweeps

Figure 3: Location of blocks

One must therefore use a different choice of rotations during alternate sweeps.

The choice of rotation angles which will ensure ordering of the computed singular values is as follows. Within blocks, the norm of the column of least index should, of course, be maximized. When rotating between blocks, the norms of the columns in the "upper" block of processor 1 should be maximized during every sweep; in the other processors, the norms of the columns in the "upper" block should be maximized during the even-numbered sweeps and those in the "lower" block should be maximized during the odd-numbered sweeps.

7 Statistical Loss of Load Balance

The most attractive feature of the one-sided Jacobi algorithm for computing the singular value decomposition on a parallel computer is the near-perfect load balance that is maintained across the processors during the computation. This, coupled with the relatively high ratio of computation to communication in the algorithm, allows the parallel algorithm to attain nearly perfect speedup over its serial counterpart.

However, as noted in Table 1 of §4 (see "Orthogonal"), as the process nears convergence an increasing number of rotations become unnecessary. Since these avoided rotations need not be distributed uniformly among the processors, the total execution time of the parallel one-sided Jacobi algorithms depends upon how much the load balance has been perturbed. In the worst case, p-1 of the processors have no rotations to perform during a sweep, while the remaining processor must perform a complete set of rotations. Despite a drastic reduction in the total work done, no decrease in parallel execution time would be realized. Conversely, in the best case, all the processors share equally in the reduction in computation, thereby reducing the total execution time as well. In general, the amount of reduction will be somewhere between these extremes. Similar remarks apply to the inner products avoided in the decoupled algorithms (see "Decoupled" in Table 1). Thus, eliminating either the unnecessary rotations or the unnecessary inner products may adversely affect the load balance, mitigating any gains that might be obtained by reducing the computation. This section presents a statistical analysis of this tradeoff.

Clearly, the rotations of pairs of columns do not form independent events. However, determining the (highly complex) correlation among the pairs of columns is problem dependent, and therefore beyond the scope of the model. Hence, we make the simplifying assumption that, given the total number of rotations performed during a sweep, each pair of columns will need rotation with equal probability. That is,

if there are 8128 total pairs of columns, of which 4000 are actually rotated during a sweep, then in the model, the probability of any pair of columns requiring rotation is 4000/8128. Comparison of the model results with the empirical results indicates that this simplification is not too extreme. We make a similar assumption regarding the probability that an inner product will need to be performed in the pairwise decoupled algorithm.

We note that our assumption is not the same as assuming that the number of rotations (resp., inner products) performed during a sweep is evenly distributed among the processors. This would be the optimum distribution (as noted above), but is unlikely to occur in practice. However, even with the above simplification, a closed-form expression for the execution time, while obtainable in principle, is too costly to evaluate. Hence, we resort to a Monte-Carlo simulation of the behavior of the algorithm. The simulation takes three input parameters: n, the number of columns in the matrix A; p, the number of processors used; and c, the number of pairs of columns that were detected as being already orthogonal (resp., the number of unneeded inner products). The output of the simulation is the expected number of parallel rotations (resp., inner products) that were avoided during the algorithm.

One iteration of the Monte-Carlo simulation is done by constructing a $p \times (N/p)$ array, where N is the total number of pairs of columns in a sweep. Each row of the array corresponds to the pairs of columns occurring in one of the processors during the sweep. The columns of this array are partitioned into blocks, representing the subsweep boundaries, when the processors must communicate. Initially, the entire array is filled with zeroes. If during the given sweep, a total of c of the N rotations (resp., inner products) were unnecessary, then c random locations in the array are set to one, representing the pairs that do not need rotation (resp., inner product). From this table, it is easy to calculate how much parallel work has been saved during that sweep. Note that simply adding up the number of ones in each row is not sufficient, since the communication done between neighboring processors after each subsweep affects the total savings, and this must be taken into account.

A series of simulations was carried out for a wide range of input parameters, and a running calculation of the mean and standard deviation of the results was made. The assumption is that the ratio of the mean to the standard deviation will asymptotically have a normal distribution. After an initial series of 30 trials (to ensure that the number of samples was large enough for the asymptotic assumption to hold), the

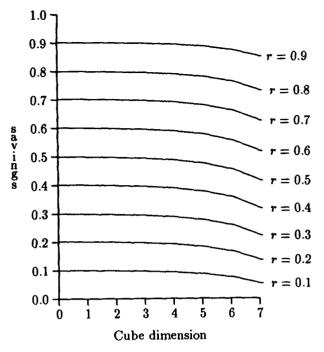


Figure 4: Savings obtained $(r := \frac{c}{N})$

model was terminated when the standard deviation, normalized by the appropriate factor to obtain a 95% confidence level, was under 0.25. Since we are interested in the number of rotations (resp., inner products) that are eliminated, the nearest integer is sufficient, permitting a relatively large tolerance. Figure 4 displays a typical graph of results for hypercubes of dimension 0 through 7, using 1024 columns and values of c that yield $c/N \approx 0.1k$ for k = 1, 2, ..., 9. Note that we compute the number of rotations (resp., inner products) avoided, so the length of the columns (i.e., number of rows m in the matrix) is not needed. However, the savings in execution time depends directly on m.

For convenience, we define r := c/N. The graph shown in Figure 4 displays the ratio of the number of parallel rotations avoided to the total number of rotations per processor, for several values of r. We note that the nature of the simulation (counting numbers rather than time) means that the same graph displays the ratio of the number of parallel inner products avoided to the total number of inner products per processor, where c now represents the total number of inner products, rather than rotations, that were avoided. From the graphs, we can see that as the number of processors increases, the percentage of the total possible savings that is actually obtained decreases. This reflects the fact that for larger numbers of processors, the potential for load imbalance is

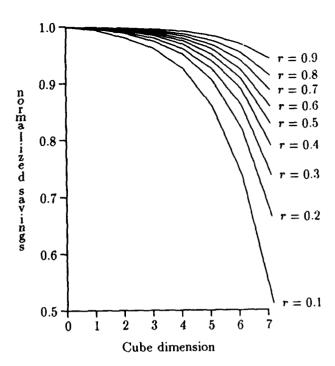


Figure 5: Normalized savings obtained $(r := \frac{\epsilon}{N})$

greater.

Figure 4 seems to indicate that the rapidity of the degradation as the number of processors increases is largely independent of r. However, Figure 4 fails to take the magnitude of the total savings into account, only the parallel savings. Let s(r, p) represent the parallel savings on p processors for a given value of r. Then Figure 4 plots s(r, p)/(N/p) against $\log_2 p$. Figure 5 uses the same data as Figure 4, but now the ordinate is $p \times s(r,p)/c$. That is, Figure 5 displays the total useful savings divided by the total savings. In the serial case all savings are useful, so this has the effect of normalizing the curves to intersect the ordinate axis at 1.0. Note that the curves corresponding to small values of c, (i.e., small values of r) fall much more rapidly than those corresponding to large values of c. This reveals that, when the number of rotations (resp., inner products) is small, increasing the number of processors severely degrades the amount of savings obtained by the modified algorithm.

The execution time of the one-sided Jacobi algorithm can be expressed as

$$T = \sum_{i=1}^{nsweeps} \left[\left(\frac{N}{p} - s_i \right) m t_f + 4 \left(\frac{N}{p} - c_i \right) m t_f + t_{comm} \right].$$

where T is the total execution time, t_f is the time for a single flop (i.e., multiply-add pair), t_{comm} is the time for the exchange of blocks between subsweeps, s_i is the number of parallel inner products saved, and c_i is

the number of parallel rotations saved. Both c_i and s_i are obtained from the Monte-Carlo simulation, given values of c and s from an actual run of the algorithm. Note that for the standard algorithm, $s_i = 0$ for all i.

Figure 6 shows a comparison of the model with the execution time of both the standard and modified algorithms for a problem of size 1600×256 with various values of p. Clearly the behavior of the algorithms is modeled with a high degree of accuracy.

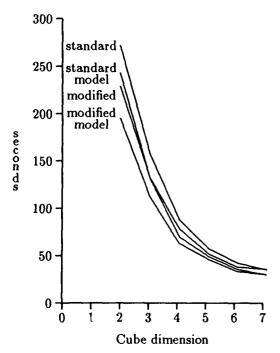


Figure 6: Comparison of models and execution times

8 Numerical Results

Parallel versions of both the standard one-sided Jacobi and the pairwise decoupled algorithm were developed on the Intel iPSC/2 and later moved to the Intel iPSC/860 at Oak Ridge National Laboratory. Both machines are hypercubes; the nodes of the iPSC/2 are based on the Intel 80386 processor, while the 860 nodes are based on the Intel i860 processor. The results we report here are for the 860, since it is faster (the clock rate is 40 MHz), has more nodes, and has more memory per node than the iPSC/2, thus allowing us to solve a wider range of problems. In fact, on a matrix of size 512×512 , 128 nodes of the 860 ran the one-sided Jacobi algorithms at over 143 Mflops. Higher rates are attainable for larger problems. Moreover, the core of the floating point calculations in the algorithm takes the form of inner products and DAXPY's (that is, $y = \alpha x + y$). These have been

Table 2: Standard (s) vs. modified (m) for n = 128

			000	1000	1 200
		m = 400	800	1200	1600
p=4	s	17.65	35.01	50.03	70.56
[m	14.88	29.71	43.59	59.97
	$\Delta\%$	15.7%	15.1%	12.9%	15.0%
8	S	9.65	19.67	31.18	39.31
	m	9.39	18.65	27.90	37.50
	$\Delta\%$	2.7%	5.2%	10.5%	4.6%
16	S	6.56	12.92	20.89	27.49
	m	6.43	12.52	18.76	24.98
	$\Delta\%$	2.0%	2.9%	10.2%	9.1%
32	s	5.11	10.57	14.58	19.38
ľ	m	5.09	9.59	14.20	18.65
	$\Delta\%$	0.4%	9.3%	2.6%	3.8%
64	s	4.70	8.18	12.00	17.44
	m	5.04	8.41	12.22	17.58
	$\Delta\%$	-7.2%	-2.8%	-1.8%	-0.8%

coded in i860 assembly language to take advantage of the floating point pipelining available on the CPU, and future versions of our codes will incorporate these routines.

The only potential drawback to using the i860 is that its ratio of communication cost to computation cost is significantly higher than for the iPSC/2. However, in the one-sided Jacobi algorithms given here, computation dominates communication provided n > 2p.

Analyzing the performance data from the parallel codes is difficult since the convergence of the one-sided Jacobi algorithm depends upon the Jacobi ordering. That is, the convergence rate depends upon the order in which the pairs of columns are rotated. However, in the parallel setting, changing the number of processors used also changes the Jacobi ordering. Therefore, one must take an average over several runs for each size matrix and number of processors to obtain meaningful results.

Tables 2 and 3 show the execution time of both the standard and modified algorithms with n=128 and n=256 (respectively) for various values of m and p. The percentage by which the modified algorithm is faster than the standard algorithm is listed beside the times. Notice that for a large number of processors, the standard algorithm is actually faster for some problems. This agrees with the analysis in the last section, since the potential savings is too small to offset the overhead of maintaining the extra data structure. However, for a small number of processors, the savings in execution time can reach as high as 20%.

				```	
		m = 400	800	1200	1600
p=4	s	71.87	135.18	202.13	272.08
	m	57.63	114.48	171.29	229.04
[$\Delta\%$	19.8%	15.3%	15.3%	15.8%
8	s	39.38	74.16	110.95	156.74
	m	33.57	66.18	98.74	132.37
<u> </u>	$\Delta\%$	14.8%	10.8%	11.0%	15.6%
16	s	22.60	44.28	66.55	87.66
	m	20.11	39.49	58.65	78.78
	$\Delta\%$	11.0%	10.8%	11.9%	10.1%
32	s	14.79	29.26	43.66	57.66
	m	14.08	27.12	39.90	52.43
	Δ%	4.8%	7.3%	8.6%	9.1%
64	s	11.45	21.68	32.23	42.50
	m	12.00	21.89	32.15	38.35
	$\Delta\%$	-4.8%	-1.0%	0.2%	9.8%
128	s	9.58	19.75	26.59	35.34
	m	10.46	21.34	27.52	36.03
	$\Delta\%$	-9.2%	-8.1%	-3.5%	-2.0%

9 Conclusions

We have shown that the standard cyclic one-sided Jacobi algorithm for the computation of the SVD of a rectangular matrix contains a significant amount of unnecessary computation. This computation takes the form of inner-products of column vectors known to be orthogonal. We have shown that a simple modification of the standard algorithm to incorporate a data structure that monitors such orthogonality (which we call "pairwise decoupling") can yield a reduction in the total execution time, despite the overhead of maintaining and updating the data structure.

The pairwise decoupled algorithm was implemented on a distributed-memory parallel computer, the Intel iPSC/860. The amount of reduction in the execution time over the standard algorithm on the 860 ranged up to 20% of the total parallel execution time. However, many of the problems exhibited less improvement than a simple calculation would indicate. This discrepancy was explained by a statistical model of the execution time of the parallel algorithm, using a Monte-Carlo simulation. Results of the simulation agreed remarkably well with the empirical results obtained from the 860.

A further modification of the Jacobi algorithm was proposed, based upon "subspace decoupling." Subspace decoupling also involves a data structure to monitor the orthogonality of pairs of columns. The difference is that in the pairwise decoupled algo-

rithm, when a pair of columns is encountered, if cither column has been modified since their previous encounter, they are assumed non-orthogonal. In the subspace decoupled algorithm, if one column of a pair to be rotated has been modified only by columns known to lie in the orthogonal complement of the other column, they are still presumed orthogonal. The potential savings in execution time for the subspace decoupled algorithm is greater than that for the pairwise decoupled algorithm; however, the communication overhead required for its implementation in a distributed-memory environment is prohibitive. We are continuing to investigate the implementation of subspace decoupling on shared-memory machines.

References

- [1] M. Berry and A. H. Sameh. An overview of parallel algorithms for the singular value and dense symmetric eigenvalue problems. J. Computat. Appl. Math., vol. 27, pages 191-213, 1989.
- [2] R. P. Brent and F. T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. SIAM J. Sci. Statist. Comput., vol. 5, pages 69-84, 1985.
- [3] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. Computer Science Department Technical Report No. 468, Courant Insitute, 1989.
- [4] P. J. Eberlein. On one-sided Jacobi methods for parallel computation. SIAM J. Alg. Disc. Meth., vol. 8, pages 790-796, 1987.
- [5] P. J. Eberlein. On using the one-sided Jacobi method on the hypercube. In M. T. Heath, editor, Proc. of the Second Conference on Hypercube Multiprocessors, pages 605-611, Society for Industrial and Applied Mathematics, Philadelphia, 1987.
- [6] P. J. Eberlein and H. Park. Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures. preprint, 1989.
- [7] F. T. Luk. Computing the singular-value decomposition on the ILLIAC IV. ACM Trans. Math. Softw., vol. 6, pages 524-539, 1980.
- [8] H. Park and P. J. Eberlein. Eigensystem computation on hypercube architectures. preprint, 1989.

- [9] W. Rath. Fast Givens rotations for orthogonal similarity transformations. *Numer. Math.*, vol. 40, pages 47-56, 1982.
- [10] P. P. M. de Rijk. A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer. SIAM J. Sci. Stat. Comp., vol. 10, pages 359-371, 1989.
- [11] C. H. Romine and K. Sigmon. Reducing inner product computation in the parallel one-sided Jacobi algorithm. *Technical Report TM-11474*, Oak Ridge National Laboratory, 1990
- [12] A. H. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer. *Math. Comp.*, vol. 25, pages 579-590, 1971.
- [13] K. Veselić and V. Hari A note on a one-sided Jacobi algorithm. Num. Math., to appear.

Basic Matrix Subprograms for Distributed Memory Systems

Anne C. Elster Cornell University School of Electrical Engineering Ithaca, New York 14853

Abstract

Parallel systems are in general complicated to utilize efficiently. As they evolve in complexity, it hence becomes increasingly more important to provide libraries and language features that can spare the users from the knowledge of low-level system details. Our effort in this direction is to develop a set of basic matrix algorithms for distributed memory systems such as the hypercube.

The goal is to be able to provide for distributed memory systems an environment similar to that which the Level-3 Basic Linear Algebra Subprograms (BLAS3) provide for the sequential and shared memory environments. These subprograms facilitate the development of efficient and portable algorithms that are rich in matrix-matrix multiplication, on which major software efforts such as LAPACK have been built.

To demonstrate the concept, some of these Level-3 algorithms are being developed on the Intel iPSC/2 hypercube. Central to this effort is the General Matrix-Matrix Multiplication routine PGEMM. The symmetric and triangular multiplications as well as, rank-2k updates (symmetric case), and the solution of triangular systems with multiple right hand sides, are also discussed.

1 Introduction

The goal of this work is to provide a set of basic "universal" matrix subprograms for the distributed memory environment that would allow programmers to implement algorithms rich in matrix-matrix operations in terms of these basic subprograms. Local communication primitives could hence be hidden in the low-level routines and the new high-level routines

not only become easier to implement, but also become portable. This has previously been done with success for serial and vector machines through the Basic Linear Algebra Subprograms (BLAS) [4,3], which among others [6] is based upon.

The high-level algorithms may not provide optimum performance measures, but our goal is to trade, say, 5-10% performance for ease of implementation and portability. Previous efforts in the same spirit include the hypercube library developed at Chr. Michelsen in Norway [2] and SCHEDULE, a parallel programming environment for FORTRAN developed at Argonne [7].

To adhere to a familiar standard, we will attempt to follow the Level-3 BLAS (BLAS3) [3] calling sequences as closely as feasible for our distributed memory case. Section 2 describes the BLAS in more detail, whereas the additional parameters needed in the distributed memory setting, follow in Section 3. The core routine, general matrix-matrix multiplication, is described in Section 4. Section 5 discusses the other BLAS routines, rank-2k updates (symmetric case), triangular multiplication, and the solution of triangular systems with multiple right hand sides, respectively. Future work and some of the issues related to the iPSC/2 implementation are mentioned in Section 6. Finally, a summary is given in Section 7.

2 The BLAS

The advantages of defining a set of basic linear algebra routines that higher-level linear algebra algorithms can be built on top of, were originally discussed by Hanson et al. back in 1979 [12]. The subprograms have later evolved through joint efforts by Dongarra et al. [5] The original routines (now dubbed Level-1 routines) limited themselves to vector-vector

operations, whereas the Level-2 routines [4] handle vector-matrix operations, and the Level-3 routines [3] explore matrix-matrix operations.

With their low number of data touches (and hence less communication needed) compared with number of arithmetic operations, the problems the BLAS3 cover, prove very suitable for distributed memory computers To follow up on this familiar standard from the sequential and shared-memory world, we have decided to follow the BLAS3 conventions for calling parameters wherever possible.

For example, the GEneral Matrix-Matrix multiplication routine in BLAS3 has the following calling format:

GEMM(TRANSA, TRANSB, M, N, K, α , A, LDA, B, LDB, β , C, LDC),

where TRANSA, TRANSB describes whether A or B transposed or not; M, N, K, the matrix dimensions; α , β , scalars; LDA, LDB, LDC, leading dimensions of A, B, C, respectively. The additional parameters needed in the distributed setting, are appended to the BLAS3 calling sequences.

3 Data Distribution and Other Calling Parameters

In the distributed memory case, extra parameters beyond the ones provided in the BLAS are needed for specifying items such as the topology of the network assumed, the data distribution desired, and possibly also parameters for indexing subclusters of processors. These parameters open up endless choices. We will, however, restrict ourselves to some of the most fundamental and useful ones. Many more options may be desirable, but too many choices defeat the purpose of having a few "core" routines that manufacturers may be willing to supply. It is the hope that sometime in the future the ideas behind these routines not only provide a standard for parallel library builders, but that optimized routines also become standard parts of future languages or operating system kernels.

Our data distribution choices are: blocksubmatrix, block-vector, and wrap-block-vector. Block-submatrix distributions facilitates orthogonal tree structures [9,8] which may be introduced to minimize communications costs compared to the more conventional distributed hypercube algorithms [16, 13]. The orthogonal structures also makes virtual transposes feasible.

The block-vector structure also maps well to hypercubes and meshes (through ring structures) and is the most common distribution of data in numerical problems. Since the individual vectors remain undistributed, it is easier to keep track of the data when doing vector oriented operations.

Finally, the wrap-block-vector mapping is considered since it provides superior load balancing in for several numerical algorithms [10,14]. As the standard block-vector approach, it is implemented using a ring structure. The extra parallel parameters (input-distr, output-distr, and network), will be added to the end of the parameter list, and the routines renamed with a P for Parallel in front of the BLAS3 name (e.g. PGEMM, for standard general matrix-matrix multiplication).

The most common and useful network topologies include hypercubes, grids (including torus), rings, and trees. This list may, however, be extended as novel architectures take on other topologies. This parameter is, perhaps, the only one that has to be modified when porting code between different architectures. Efficiency of the code will, however, be somewhat linked to the data structures (though the communication bandwidth of the system is probably of more importance). For instance, true ring topologies do not emulate grid structures, broadcast, and gather, as efficiently as, say, hypercubes.

Useful communication structures include rings, trees, and meshes. Rings are commonly used in numerical algorithms where operations are performed on block-vectors. They may be embedded on a hypercube network using all nodes by numbering the processors according to 1-D Gray codes. [17,1,9]. The Gray codes ensure that processors that are next to each other in the ring structure also achieves nearneighbor communication between physical hypercube nodes. This embedding also includes a spanning tree (Figure 1).

Meshes (including toroidal connections) may similarly be embedded on hypercubes using 2-D Gray codes. This embedding includes a set of orthogonal tree structures [9]. Whereas tree structures provide efficient structures for broadcast and gather

(both processor-row/column-wise and network-wide), grids - perhaps the most common parallel network - are well-suited for block-submatrix data distributions which are common in applications such as image ans seismic processing.

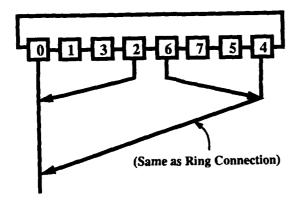


Figure 1. Ring embedding hypercube using binary reflective Gray code. Tree structure for broadcast/gather also shown.

4 Parallel Matrix-Matrix Multiplication

The general matrix-matrix multiplication routines (GEMMs) are the core of the BLAS-3 library. For real matrices A, B, and C (α and β are scalars), the operations can be described as follows:

 $C \leftarrow \alpha AB + \beta C$, where where A and/or B may be transposed.

The scalar multiplication (α, β) may simply be performed by broadcasting the scalar value(s) to each node and then perform the scaling locally. We shall, however, assume that the scalars α and $\beta=1$ in our discussion for simplicity. Their computation will also not be affected by the data distribution (block-column or submatrix-block). A discussion of the matrix product AB follows. Note, that the matrix addition included in the matrix operations above, gets performed along with the multiplication as the result gets added in during the accumulation of the summation of $\sum a_{ik}b_{kj}$.

Of the four BLAS permutations allowed through the TRANSA and TRANSB options (see last section), we will first take a closer look at the AB case. On a torus, computing the the products $a_{ik}b_{kj}$ using block-submatrix distributions, can be achieved by rotating the distributed B matrix east-west through the processor plane as the appropriate data reaches the processors. Orthogonal structures may then be used to gather the summations. These structures will also be used for the $A^T B^T$, $A^T B$ and AB^T cases.

Similarly, in a ring setting, whole block-vectors are rotated left-right on a ring instead of the submatrices for a mesh in the summation phase. Notice that the ring structure mapping also provides a binary tree structure when implemented using the binary reflective Gray code. This allows for efficient broadcasts and gatherings of data.

For A^TB and AB^T , which matrix (A or B) to rotate through the processors in order to avoid stride problems, depends on the storage convention of the matrices. Finally, in the A^TB^T case, the data needs to be "transposed" in order to compute the products $a_{ik}b_{kj}$. The data may also need to be reordered locally to avoid stride problems.

If one of the matrices A or B is symmetric, then, either $A = A^T$ or $B = B^T$. These cases can hence be viewed as the A^TB and AB^T cases described in the previous section. We are here assuming that compressing the storage of symmetric matrices is not worth while a the distributed memory setting. Although more costly in storage, the cost in increased algorithmic complexity seems to outweight the benefits. Also, in the block-submatrix case one would not be able to take full advantage of the orthogonality of the hypercube structure if storage compression is used.

5 Other BLAS Routines

Following a brief discussion of how the distributed memory setting will affect the rest of the BLAS routines.

5.1 Rank-2k Updates of a Symmetric Matrix

We here consider the following updates of a symmetric matrix C covered by the BLAS3:

$$\mathbf{C} \leftarrow \alpha \mathbf{A} \mathbf{B}^{\mathbf{T}} + \alpha \mathbf{B} \mathbf{A}^{\mathbf{T}} + \beta \mathbf{C}$$
$$\mathbf{C} \leftarrow \alpha \mathbf{A}^{\mathbf{T}} \mathbf{B} + \alpha \mathbf{B}^{\mathbf{T}} \mathbf{A} + \beta \mathbf{C}$$

The Rank-1k cases are covered by the general matrix-matrix multiplication routines. In the case of the rank-2k updates of symmetric matrices, all matrix-matrix products are of the form A^TB or AB^T , which, as mentioned, does not require transpositions. Notice that since $AB^T = (BA^T)^T$ and $A^TB = (B^TA)^T$, only one of the products needs to be computed and the remainder of the computation reduces to matrix additions (with scaling with α and β).

5.2 Triangular Matrix Multiplication

We here consider permutations of multiplying the dense matrix B with a triangular matrix T:

 $\mathbf{B} \leftarrow \alpha \mathbf{TB}$, where T and/or B may be transposed.

Notice that if one considers T upper triangular, then the T^T case would represent the lower triangular case and vice versa.

Triangular matrix multiplication may easily be performed redistributing only data from T. In the ring/block-vector case, the two first multiplications (TB and TTB) may be computed rather straightforwardly (with respect to communication) since the B matrix already is distributed in the same block-column fashion as used for the general multiplication case. For BT and BTT, however, the matrix B is distributed in a block-column fashion whereas the general method assumes a row-wise access. In this case, redistributing T would hence not be sufficient.

5.3 Triangular Systems with Multiple Right-Hand Sides

In this section, orthogonal data structures are introduced in the context of solving some basic linear systems. First, triangular systems with multiple righthand sides will be considered:

$$B \leftarrow \alpha T^{-1}B$$

$$B \leftarrow \alpha T^{-T}B$$

$$B \leftarrow \alpha B T^{-1}$$

$$B \leftarrow \alpha B T^{-T}$$

Here α is a scalar, $\mathbf{B} \in \Re^{m \times n}$, and \mathbf{T} a non-singular triangular matrix. Notice how both the inverse (T^{-1}) and inverse transpose (T^{-T}) cases are considered for \mathbf{T} providing both the upper and the lower triangular cases.

Since triangular solves involve either forward or backward substitution (both inherently sequential operation), parallelization is not as straight-forward as in the multiplication cases. However, decent parallelization can be achieved by using a "pipelined" approach, as described by [14], where the data is mapped to a ring structure. In this case, a wrapblock-vector data distribution since it provides a better processor utilization in the factorization stage [10].

It was recently shown that these other BLAS-3 subprograms can actually be implemented in terms of GEMM, at least in the sequential setting [15] with reasonable efficiency. This would be desirable in the parallel setting as well, since it would reduce the machine-depended encodings to that of PGEMM. Futher investigations of this idea are currently under considerations.

6 Some Implementation Issues and Future Work

The Intel iPSC/2 hypercube is currently being used as a test-bed for implementing the PBLAS routines. The ideas behind the routines are not ment to limit themselves to the Intel cube or its topology, but the Intel machine is rather used as test environment for how the PBLAS may be developed for common distributed memory systems. As mentioned, It is our hope that the PBLAS can become a standard for how core matrix algorithms are developed.

We chose to implement our routines in C. Fortan may seem, to many, the most natural language to implement matrix algorithms in. However, C, with its powerful pointer constructs for dynamic memory allocation and strong link to UNIX, is rapidly becoming more popular. Since we wanted to use some of the C pointer features in the implementation, it became a natural choice. C also interfaces well with Fortran and is along with Fortran 77 available in the Intel hypercube. (Fortran 90 may provide similar features, but is yet not available for the Intel cube.)

At the present, PGEMM has been partially imple-

mented. Current work includes completing most of C_me = C_me + A_me * B_me,me the PGEMM cases (transpose of A and B, various data distributions, etc - see Section 3), benchmarking it, and developing some examples of application. Future work includes the implementation of some of the other PBLAS routines cases (quite probably with call to PGEMM). Since out goal is to demonstrate the concepts rather than provide production code, we will, for now limit ourselves to a couple of core cases rather than provide a full PBLAS implementation for the Intel hypercube (left as future work for people providing production code).

6.1 PGEMM on the hypercube

Taking a closer look at the case $C \leftarrow C + A \cdot B$ in a ring setting, we decided to store the matrices on the nodes in one-dimensional arrays. These are then used directly as message-buffers during the communication phases saving valuable storage space and copy-time. To emulate 2-dimensional array index, index functions where defined: including the leading dimension of the respective matrix (LDX):

Indexing function for A, B and C: This indexing function assumes matrices stored-bycolumn starting at array location A[1] (FORTRANtype).

The block-column version of $C \leftarrow C + A \cdot B$ can be described by the following equation for block-vector C_i :

$$C_i = A_1B_{1i} + A_2B_{21} + ... + A_pB_{pi}$$
, for $i, j = 1: p$, where p is the number of block-vectors $(n = r \cdot p)$, where r is block-width).

Assuming a ring embedding using the Binary Reflective Gray Code [17], the above equation leads to the algorithm:

PGEMM

Let me = position in ring

(As in the equation above where node i holds C_i and Ai locally. Also reflects which part of the matrices are stored locally.)

Compute on local data:

Following the MATLAB notation as described in Golub-Van Loan [11]) we here have (all local subblocks):

```
C_{me} = C[1:n,(me-1)r:me * r]
A_me = A[1:n,(me-1)r:me * r]
B_{me,me} = B[(me-1)r:me + r,1:r]
```

(numnode = no. of nodes in cube (ring))

 $A_{tmp} = A_{me}$

```
FOR p = 1 to numnodes
  SEND A_tmp to left-neighb
  RCV A_tmp from right-neighb
  C_me = C_me + A_tmp*B_x,me
```

(x is a function of numnodes and p corresponding to the above equation)

Leave result on nodes or SEND to host.

END{PGEMM}

Other cases will be described in future work along with a discussion on how to access submatrices (leaving some processor idle rather than redistributing data).

Conclusions

In this paper, a basic set of linear algebra algorithms in the spirit of BLAS [4,3] were proposed to form a basis for algorithmic development also in the distributed memory environment.

Extra parameters needed for the parallel environment were identified and added to the BLAS3 calling sequences. These parameters included a parameter to describe the network topology and parameters for specifying the input and output distribution of the data. Powerful communication structures (such as the orthogonal data structures involving trees and meshes) could then be hidden within the routines sparing the users from hardware details.

It is the hope that sometime in the future the ideas behind these routines not only will provide a standard for parallel library builders, but that similar optimized routines also become standard parts of future languages or operating system kernels.

Acknowledgments

The author would like to thank Charles F. Van Loan, her thesis advisor, for his helpful suggestions and support. Thanks are also due the Cornell Theory Center for providing access to their 32-node Intel IPSC/2 hypercube system.

This research is supported in part by the U.S. Army Research Office through the Mathematical Sciences Institute, Cornell University.

References

- R.M. Chamberlain. Gray codes, Fast Fourier Transforms and Hypercubes. Technical Report 864502-1, Chr. Michelsen Institute, Bergen, Norway, May 1986.
- [2] R.M. Chamberlain, P.O. Frederickson, J. Lindheim, and J. Petersen. A High-Level Library for Hypercubes. Hypercube Multiprocessors 1987, pages 651-655, 1987.
- [3] J.J. Dongarra, J. Du Croz, I.S. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report AERE R 13297, Harwell Laboratory, Oxfordshire, England, October 1988. Also ANL:TM 88 and NaG:TR 14/88.
- [4] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. ACM Trans. on Mathematical Software, 14:1-17, 1988.
- [5] J.J. Dongarra and S.J. Hammarling. Evolution of Numerical Software for Dense Linear Algebra. In M.G. Cox and S.J. Hammarling, editors, Advances in Reliable Numerical Computation, Norfolk, VA. – to be published.
- [6] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. LINPACK User's Guide. SIAM, Philadelphia, 1979.
- [7] J.J. Dongarra and D.C. Sorensen. A Portable Environment for Developing Parallel FORTRAN programs. Parallel Computing, 5(1&2):175-186, July 1987.
- [8] A.C. Elster and H. Li. Hypercube Algorithms on the Polymorphic Torus. In G. Fox, editor,

- The Fourth Conference on Hypercubes, Concurrent Computers, and Applications. AMC, March 1989. to appear.
- [9] A.C. Elster and A.P. Reeves. Block-matrix Operations Using Orthogonal Trees. In G. Fox, editor, The Third Conference on Hypercube Concurrent Computers and Applications, pages 1554-1561, Pasadena, CA, January 1988. ACM. Vol. II.
- [10] G.A. Geist and M.T. Heath. Matrix Factorization on a Hypercube Multiprocessor. Hypercube Multiprocessors 1986, pages 161-180, 1986.
- [11] G.H. Golub and C.F. Van Loan. Matrix Computations. Johns Hopkins, Baltimore, MD, second edition, 1989.
- [12] R. Hanson, D. Kincaid, F. Krogh, and C. Lawson. Basic Linear Algebra Subprograms for Fortan Usage. ACM Transaction on Math. Software, 5:153-165, 1979.
- [13] S.L. Johnsson. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. Journal of Parallel and Distributed Computing, 4:133-172, 1987.
- [14] G. Li and T.F. Coleman. A New Method for Solving Triangular Systems on a Distributed Memory Message-Passing Multiprocessor. SIAM Journal of Sci. Stat. Comp., 10(2):382-396, March 1989.
- [15] C.F. Van Loan and B. Kågström. Poor-Man's BLAS for Shared Memory Systems. preliminary version through personal communications.
- [16] O.A. McBryan and E.F. Van de Velde. Matrix and Vector Operations on Hypercube Parallel Processors. *Parallel Computing*, 5(1 & 2):117-125, July 1987.
- [17] J. Salmon. Binary Gray Codes and the Mapping of Physical Lattice into a Hypercube. Caltech Concurrent Processor Report Hm-51, Caltech, 1983.

Linear Algebra for Dense Matrices on a Hypercube *

Mark P. Sears

Sandia National Laboratories Albuquerque, NM 87185

Abstract

A set of routines has been written for dense matrix operations optimized for the NCUBE/6400 parallel processor. This work was motivated by a Sandia effort to parallelize certain electronic structure calculations [1]. Routines are included for matrix transpose, multiply, Cholesky decomposition, triangular inversion, and Householder tridiagonalization. The library is written in C and is callable from Fortran. Matrices up to order 1600 can be handled on 128 processors. For each operation, the algorithm used is presented along with typical timings and estimates of performance. Performance for order 1600 on 128 processors varies from 42 MFLOPs (Householder tridiagonalization, triangular inverse) up to 126 MFLOPs (matrix multiply). We also present performance results for communications and basic linear algebra operations (saxpy and dot products).

Introduction.

This paper describes the implementation of routines for dense linear algebra on the NCUBE/6400 hypercube. The primary purpose of these routines is for electronic structure calculations, and emphasis has been placed on routines for the generalized symmetric eigenvalue problem, although we intend to add routines for linear solutions and SVD.

A coarse-grain MIMD machine such as the NCUBE might not be expected to perform particularly well on linear algebra problems when compared with SIMD machines (e.g. the Connection Machine) or vector machines like the CRAY-XMP and its successors. However, electronic structure programs spend most of their time computing matrix elements, a task which is very well suited to the MIMD architecture. In order to retain this advantage, it is important to have a library of linear

algebra software with good performance. This was the object of the work presented here.

The NCUBE/6400 hypercube can be configured with up to 8192 processors (nodes). Sandia currently has two of these systems, one with 8 nodes and 4MB of memory per node and the other with 128 nodes with 1MB of memory per node. The latter system is being upgraded to 1024 nodes with 4MB per node. The NCUBE/6400 uses a second generation NCUBE CPU chip that has some architectural differences with the NCUBE/ten CPU chips. The measured floating point and communications parameters of the new NCUBE are described below. We use the notation NCUBE-II to refer to this processor with the initial software release, clock cycle time (20MHz), and memory wait states. Later versions of the 'NCUBE/6400' might be released with rather different properties. We also use the notation NCUBE-I to refer to processors of the NCUBE/ten and other first generation NCUBE systems.

In the following, we present results for the basic floating point and communications performance of the NCUBE-II. We then describe the basic operations such as transpose, mapping, and matrix multiply. The performance of the Cholesky factorization and inverse of a triangular matrix are presented next, followed by the results for Householder tridiagonalization.

NCUBZ-II floating point and communications parameters.

Linear algebra software relies heavily on a few simple kernels [2] and it is important to optimize these carefully for a particular machine architecture. The NCUBE-II has a simple memory hierarchy [4] with only main memory and an instruction cache. There are no vector instructions or vector registers. The C compiler is capable of using only a small number of registers for inner loops and only if explicitly assigned by the user. We therefore chose not to implement any level-2 or level-3 BLAS operations.

Table 1 shows timing results for dot product and

^{*}This work was performed at Sandia National Laboratories which is operated for the U.S. Department of Energy under contract number DE-AC04-76DP00789.

Table 1: Measured floating point times for dot products and saxpy operations on a single NCUBE-II processor. Times are in microseconds per step for a vector length of 100. The labels r4,r8 refer to single precision real, double precision real, and c4,c8 refer to single precision complex, double precision complex. Single precision is 4 byte IEEE floating point and double precision is 8 byte IEEE floating point.

Function	Time/step	MFLOPs
dot,r4	1.66	1.20
dot,r8	1.74	1.15
dot,c4	6.13	1.31
dot,c8	5.21	1.54
saxpy,r4	1.77	1.13
saxpy,r8	1.66	1.20
saxpy,c4	7.06	1.13
saxpy,c8	6.26	1.28

saxpy operations. The single precision dot product routine accumulates the result in double precision, and requires 2 flops and 2 memory references per loop step, plus a single to double conversion. The saxpy operation requires 2 flops and 3 memory references per step. The saxpy operations are therefore a little faster than one might expect based on the number of memory references. The complex versions of these operations require 8 flops per step. Using these loops we can define a floating point operation time τ_f which is an average of the dot or saxpy times.

The kernel routines are written in C, with several optimizations which contribute significantly to the speed. Registers are carefully assigned to loop indices and pointers. Pointer arithmetic is used rather than array references. The loops are decremented (n to 1) rather than incremented. There is some loop unrolling. These optimizations generally give a speedup of 2-3 times that of naive code. We estimate that future compiler optimimizations and/or assembly coding of these routines may give improvements of 20%-50%. We note that these C routines are about 12 times faster than hand coded assembler versions written for the NCUBE-I.

We have also measured the communication parameters τ_o for startup time and τ_c for transfer time per byte. This is done with a program which runs on a pair of nodes which exchange a large number of messages. Different tests varied the length l of the messages from 1 to 1000 bytes and the results were fitted to an expression of the form T=a+lb. Deviations from the straight line fit were about 5%. The

Table 2: NCUBE-II floating point and communications parameters. All times are in microseconds.

Single precision flop time τ_f	.84
Double precision flop time τ_f	.77
Message startup time, τ_o	151
Message time/byte, τ_c	.37

measurement does not distinguish between times for message read and message write, so we take $\tau_o = a/2$ and $\tau_c = b/2$.

Table 2 summarizes the measured floating point τ_f and communication parameters for the NCUBE-II. Note the referred range value τ_o relative to τ_f or τ_c . Independent measurements using Fortran give $\tau_o = 200$ and $\tau_c = .4$ [5].

Matrix organization.

All of the routines described here allocate entire columns of the matrix to the processors. Routines are provided to map between different matrix organizations (columns blocked or wrapped, and either binary or Gray-code imbedding). Thus, algorithms which use any of these organizations can be implemented.

An alternative to column organization is mesh organization, where the matrix is imbedded in a 2D mesh of processors. Routines have been written to map between column and mesh organizations, but none of the algorithms presented here use mesh organizations.

Basic matrix routines.

Basic routines include matrix setup, mapping routines, matrix transpose, matrix norms, and user functions (i.e., apply a user defined function to all elements of the matrix). Timings for the mapping routines are shown in Table 3 for a matrix of size 1500 on 64 processors. These routines have a typical complexity of $n^2 \log P/P$ where n is the matrix order and P is the number of processors [6].

The matrix transpose has a complexity similar to the above routines. Times are .48 seconds for n = 1024, P = 64 and 1.04 seconds for n = 1500, P = 64.

Matrix multiply.

Matrix multiply is implemented as a matrix transpose followed by column-wise dot products. The

Table 3: Timings in seconds for mapping routines. N refers to natural or binary processor ordering, G refers to Gray-code processor ordering, B refers to columns blocked, and W refers to columns wrapped. Timings are for 64 processors and matrix size 1500 by 1500.

[N, B	N, W	G, B	G, W
N, B	-	.42	.041	.8
N, W	.42	-	.78	.001
G, B	.041	.78	-	1.2
G, W	.8	.001	1.2	-

Table 4: Timings in seconds for matrix multiply. MFLOP rates follow in parentheses and are computed from the serial complexity of the algorithm, $2n^3$ flops.

N	P=64	P=128
128	.15 (27)	.11 (36)
256	.72 (44)	.45 (71)
512	4.70 (57)	2.63 (102)
1024	32.9 (65)	17.37 (123)
1300	68.0 (65)	37.42 (117)
1600		64.8 (126)

multiply algorithm uses a (Gray,block) organization and calls the mapping routine if necessary. The left hand factor is transposed and cycled around the imbedded ring of processors. Table 4 shows timings and MFLOP rates for P=64 and P=128 for various matrix sizes.

Cholesky factorization and triangular inverse.

The Cholesky factorization algorithm is similar to that of Geist, et al. [8]. The algorithm implemented here computes the factorization $A = U^T U$ where U is upper triangular; it is therefore a column version of Geist's row algorithm. The (Gray, wrap) mapping is used.

In contrast to Cholesky factorization, the algorithm to compute the inverse of a triangular matrix uses a (Gray,block) mapping. The wrap version of this algorithm incurs very large communication costs. Even the block version must be modified to prevent the message traffic along the imbedded ring from clogging up. The NCUBE software provides buffered asynchronous communications, and the tri-

Table 5: Timings in seconds for Cholesky factorization and triangular inverse. MFLOPs follow in parentheses and are computed from the serial complexity of the algorithms, $n^3/3$ in both cases.

Cholesky factorization.

N	P=64	P=128
128	.15 (4.7)	.14 (5.0)
256	.46 (12.)	.40 (14.)
512	1.87 (24)	1.45 (30)
1024	9.38 (38)	6.45 (56)
1300	17.2 (43)	11.4 (64.2)
1600		18.7 (73)

Triangular inverse.

N	P=64	P=128
128	.10 (7.0)	.09 (7.8)
256	.41 (14)	.27 (21)
512	2.37 (19)	1.38 (32)
1024	16.25 (22)	8.81 (41)
1300	33.4 (22)	18.5 (39.5)
1600	<u>[-</u>	32.4 (42)

angular inverse algorithm begins at the last column by computing a small amount and then sending a large column to the left. Therefore, without sychronization, the buffer memory of a node down the ring quickly fills up and the software does not recover from this state. A simple fix is to synchronize each stage of the ring after some number of bytes (chosen to be a fraction of the available buffer memory) have been passed.

Table 5 summarizes the timing results for Cholesky factorization and triangular inverse.

Householder tridiagonalization.

Unlike Householder orthogonalization, Householder tridiagonalization cannot be pipelined [7]. This forces the use of broadcast communications. Moreover, there is a significant serial component to the algorithm. The algorithm proceeds as follows. The matrix is distributed using the (natural, block) mapping. For c = 1 to n - 2 we then execute the following:

1. On the processor containing column c, compute the Householder vector u. This requires about 3i flops, where i = n - c - 1, so the time for step 1 is $T_1 = \frac{3}{2}n\tau_f$. This is part of the serial overhead.

- 2. Broadcast u to all processors. This takes time $2\log(P)(\tau_o + 4i\tau_c)$ for each step, so $T_2 = 2n\log(P)(\tau_o + 2n\tau_c)$.
- 3. Compute w = Au in parallel. Each processor computes its share of w using its share of the columns of A. The total time for this step is $T_3 = \frac{2}{3}n^3\tau_f/P$.
- 4. Compute the inner product of u with u and u with w. The calculation time for this is small, but the results must be exchanged among all processors. The time is therefore $T_4 = 2n \log(P)\tau_o$.
- 5. Combine $q = \alpha u + \beta w$ where α and β are computed from the inner products above. The calculation is done in parallel and is small, but the resulting q must be combined on all nodes. The time for this is $T_5 = 2n \log(P)(\tau_o + 2n\tau_c)$.
- 6. Compute $A^{(c+1)} = A^{(c)} + uq^T + qu^T$ in parallel. The time for this is $T_6 = \frac{4}{3}n^3\tau_f/P$ if no advantage is taken of symmetry.

Summing the contributions for n-2 columns, the total time is

$$T = (2\tau_f/P)n^3 + (\frac{3}{2}\tau_f + 8\log(P)\tau_c)n^2 + (6\log(P)\tau_o)n^2$$

for single precision floating point, where we have kept only the leading terms. Using this expression and the parameters from Table 2, we obtain 11.36 seconds for n = 512, P = 64 and 54.6 seconds for n = 1024, P = 64. The actual results are shown in Table 6; agreement is good. The biggest part of the overhead is due to communications. A fairly simple modification of the above algorithm reduces the terms proportional to $\log(P)$ by a factor of 2. This is done by running the algorithm backwards from c = n to 2 and collapsing the broadcast and combine operations to smaller subcubes as the calculation progresses. This modification has been tested on the NCUBE-I but has not yet been implemented on the NCUBE-II.

Conclusions.

This paper has presented algorithms and performance for a library of dense matrix linear algebra routines. The library is optimized for the NCUBE/6400, a new-generation MIMD hypercube. Performance for dot products and saxpy operations varies from 1.1 to 1.5 MFLOPs on a single node.

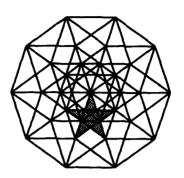
Table 6: Timings in seconds for Householder tridiagonalization. MFLOPs follow in parentheses and are computed from the serial complexity of the algorithm, $\frac{4}{3}n^3$ flops.

N	P=64	P=128
128	.414 (6.7)	1.06 (2.6)
256	2.88 (7.3)	2.80 (7.5)
512	11.47 (16)	9.28 (19)
1024	61.8 (23)	41. (35)
1600	-	128. (43)

The results for up to 128 processors are encouraging: for n = 1600 we get performance varying from 42 MFLOPs (Householder tridiagonalization, triangular inverse) up to 126 MFLOPs (matrix multiply). These results are encouraging for our electronic structure calculations.

References

- P. Feibelman, Efficient solution of Poisson's equation in linear combination of atomic orbital (LCAO) electronic structure calculations. J. Chem. Phys <u>81</u>,12, pp 5864-5872.
- [2] C. Lawson, R. Hanson, D. Kincaid, F. Krogh, Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Soft. 5,3, pp 308-323.
- [3] G. Golub, C. Van Loan, Matrix Computations, (Johns Hopkins University Press, Baltimore, MD, 1989).
- [4] The NCUBE 6400 Processor Manual, NCUBE, Beaverton, OR, 1989.
- [5] S. Plimpton, private communication.
- [6] C. Ho and S. Lennart Johnsson, Matrix Transposition on Boolean n-cube Configured Ensemble Architectures. Yale University report YALEU/DCS/TR-494 1986.
- [7] J. Dongarra and D. Sorensen, A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem.
- [8] G. Geist and M. Heath, Matrix Factorization on a Hypercube Multiprocessor. Hypercube Multiprocessors 1986, pp 161-180.



The Fifth Distributed Memory Computing Conference

12: Sparse Matrix Algorithms

Incremental Condition Estimator for Parallel Sparse Matrix Factorizations

Jesse L. Barlow* and Udaya B. Vemulapati†
Department of Computer Science
The Pennsylvania State University
University Park, PA 16802.

Abstract

There is often a trade-off between preserving sparsity and numerical stability in sparse matrix factorizations. In applications like the direct solution of Equality Constrained Least Squares problem, the accurate detection of the rank of a large and sparse constraint matrix is a key issue. Column pivoting is not suitable for distributed memory machines because it forces the program into a lock-step mode, preventing any overlapping of computations. So factorization algorithms on such machines need to use a reliable, yet inexpensive incremental condition estimator to decide on which columns to be included. We describe an incremental condition estimator that can be used during a sparse QR factorization. We show that it is quite reliable and is well suited for use on parallel machines. We supply experimental results to support its effectiveness as well as suitability for parallel architectures.

1. Introduction

Choosing a set of linearly independent columns from a given matrix, within a tolerance of machine precision, is a common subproblem, among problems involving matrix computations. Subtle variations of the same problem are "rank detection" and "condition estimation".

Traditional methods of rank detection for dense matrices include QR factorization with column pivoting [9], Singular Value Decomposition [9] and a host of condition estimators [11], the most popular among them being the LINPACK 1-norm estimator. Threshold pivoting [10] strategy is often used in the case of sparse matrices.

However, when we consider solving these problems on a parallel architectures, most of the traditional approaches fail to be cost effective, especially when large and sparse matrices are involved.

In this paper, we propose an incremental condition estimator, which is quite reliable and is well suited for parallel sparse matrix QR factorizations. In section 2, we examine the reasons for the failure of traditional methods when applied to our problem. In section 3 we discuss the issues in the effective implementation of solving our problem on a parallel architecture. In section 4, we describe the an algorithm that allows us to incrementally estimate the condition number of the triangular factor during the factorization. In section 5, we discuss the implementation issues on a parallel architecture and provide experimental results. In section 6, we provide experimental evidence that suggests that the algorithm is robust enough.

2. Failure of Traditional Methods

The general strategy for doing a QR factorization of a sparse matrix C is [6]

- 1. Determine the symbolic structure of C^TC .
- 2. Using a heuristic approach, find a permutation matrix P, such that P^TC^TCP has a sparse cholesky factor.
- 3. Generate the storage structure for R by doing a symbolic factorization of P^TC^TCP .
- 4. Compute R numerically.

Although it is known that finding a permutation in step 2, that produces an optimally sparse Cholesky factor is a hard problem (in fact NP-hard), many good heuristic approaches such as minimum degree and nested dissection give us fill-reducing orderings [7]. This approach of determining the data structures required for the R factor before the actual factorization (in other words a static data structure) has some

^{*}Research supported by the National Science Foundation under grant no. CCR-8700172, the Air Force Office of Scientific Research under grant no. AFOSR-88-0161, and the Office of Naval Research under grant no. N0014-80-0517.

[†]Research supported by the Office of Naval Research under grant no. N00024-85-C-6041.

advantages, compared to dynamically set up storage structures during the factorization. The accessing of the elements in a static set up is faster and hence the factorization step is likely to be faster. Since the static structure does not depend on the numerical values of the original matrix C, the cost involved in steps 1-3 can be spread over a number of factorizations if repeated computations of R are required with different numerical values of C.

Most of the known algorithms for rank detection (or condition estimation) are neither cost effective nor appropriate for sparse matrix applications. Most of the estimators, surveyed by Higham [11], require $O(n^2)$ units of computation time. This is too expensive for sparse matrices, considering that the factorization of a sparse matrix itself requires only $O(n^{1.5})$. The QR factorization with column pivoting upsets the sparsity pattern, because the column ordering chosen in step 2 is not used. Moreover the pivoting process requires us to use a dynamic data structure for R. Singular Value Decomposition is too expensive for practical use, even though it is the most accurate algorithm for rank detection.

3. Issues in Parallel Factorization

In this paper, we limit our discussion to distributed memory machines, such as Hypercubes, while talking about parallel architectures. If we want to implement the factorization in parallel, we need to re-examine the validity of the traditional methods on such machines. The column pivoting algorithm requires that the processors have to synchronize to select the next pivot column. This introduces not only delays due to communication overheads but also forces the program into a lock-step mode, leaving no room for pipelining and / or overlapping of computations. As was observed already, any pivoting process results in more fill-in and hence more computation time.

Dynamic data structures are not easy to distribute in a local-memory environment; even if we manage to do that, keeping track of the current state of the structure among all processors is not an easy task. Hence we consider using static data structures. The threshold strategy described by Heath [10] and implemented in SPARSPAK-B [8] allows to deal with the static data structures for most of the computations. Even though empirical tests show that this strategy rarely fails in practice, dramatic failures in rank detection are possible in some cases. A simple example is the following bidiagonal matrix, which will be considered full rank matrix for any value of a, even though D could be arbitrarily ill-conditioned. (In

fact, actual $\kappa(D) \approx a^3$).

$$D = \begin{pmatrix} 1 & a & 0 & \dots & 0 \\ 0 & 1 & a & \dots & 0 \\ 0 & 0 & \ddots & a & 0 \\ 0 & \dots & & 1 & a \\ 0 & \dots & & 0 & 1 \end{pmatrix}$$

If we use static data structures, during the factorization, we are only allowed to look at each column only once in a given sequence and we should be able to determine whether a new column is linearly independent of the others already chosen to be in the factor. This translates to checking whether the resulting upper triangular factor is going to be well conditioned.

To this end, Bischof [3] describes an incremental estimator for the smallest singular value, which is a modified 2-norm condition estimator suggested by Cline, Conn and Van Loan [4,13]. However this algorithm has two serious drawbacks when it comes to sparse matrices. Firstly the estimator requires n^2 flops during the triangularization of an $n \times n$ matrix. Secondly, its estimate of the smallest singular value differs arbitrarily from the actual value, for matrices with special structure. In particular, if the new row being added is orthogonal to the current approximate singular vector, then the estimate is likely to be very poor. As an example, consider the following 3×3 matrix.

$$D = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \beta & -\beta\omega & 2 \end{array} \right].$$

For a specific value of $\omega = 1 - \sqrt{2}$, the estimate of the smallest singular value from Bischof's algorithm is 1, independent of β , while the actual 2-norm of $D \approx \beta^2$. We are able to construct such trivial 3×3 examples, because there is no look-ahead in the estimation algorithm.

Recently Bischof, Lewis and Pierce [2] extended the original algorithm to handle the case of general matrices and showed how this modified approach can be used for nested dissection case.

4. Incremental Condition Estimation

The proposed algorithm is an "incremental ∞ -norm" estimator that uses look-ahead. It is a modification of LINPACK 1-norm estimator for upper triangular matrices [5]. The algorithm looks at each column just once and decides whether to include that column in the factorization or not. It does that by incrementally estimating the ∞ -norm of the inverse of the partially

formed upper triangular factor. There is no column pivoting and hence static data structures can be used.

We are interested in computing a QR factorization of the matrix C, with accurate rank detection, so that the factored matrix has the following form

$$C = Q \left[\begin{array}{cc} U_{11} & U_{12} \\ 0 & U_{22} \end{array} \right].$$

Define the sequence of upper triangular matrices $U_1^{(k)}, k = 1, 2, ..., l$, by

$$U_1^{(1)} = (u_{11})$$
 where $u_{11} = ||c_1||_2$

and

$$U_1^{(k+1)} = \left[\begin{array}{cc} U_1^{(k)} & v_{(k+1)} \\ 0 & \gamma_{k+1} \end{array} \right]$$

where

$$v_{k+1} = (c_{1,k+1}^{(k)}, \dots, c_{k,k+1}^{(k)})^T,$$

$$c_{k+1}^{(k)} = (c_{1,k+1}^{(k)}, \dots, c_{m,k+1}^{(k)})^T$$

is the $(k+1)^{st}$ column of C after H_1, H_2, \ldots, H_k are applied and

$$\gamma_{k+1} = \|(c_{k+1}^{(k)}, \dots, c_{m_1, k+1}^{(k)})^T\|_2$$
$$= \sqrt{\|c_{k+1}^{(k)}\|_2^2 - \|v_{k+1}\|_2^2}$$

Let

$$L^{(k)} = [U_1^{(k)}]^T$$

and

$$a^{(1)} = 1;$$

$$\hat{\sigma}_1 = 1/u_{11} = 1/||c_1||_2 = 1/\gamma_1.$$

To choose the $(k+1)^{st}$ column, we let $x^{(k)}$ be such that

$$L^{(k)}x^{(k)} = a^{(k)}$$

where $a^{(k)}$ is a vector of ± 1 , chosen to maximize $||x^{(k+1)}||_{\infty}$. Then compute

$$x^{(k+1)} = (x^{(k)}, \xi_{k+1})^T$$

where

$$\xi_{k+1} = \gamma_{k+1}^{-1}(-sgn(v_{k+1}^Tx^{(k)}) - v_{k+1}^Tx^{(k)}).$$

Thus

$$\hat{\sigma}_{k+1} = \max\{\hat{\sigma}_k, |\xi_{k+1}|\} = ||x^{(k+1)}||_{\infty}.$$

This procedure is precisely the LINPACK estimator without the "look-ahead" property.

To incorporate a "look-ahead", we consider the partial sums

$$\rho_j^+ = v_j^T \left[\begin{array}{c} x^{(k)} \\ \xi_{k+1}^+ \end{array} \right] \text{ and } \rho_j^- = v_j^T \left[\begin{array}{c} x^{(k)} \\ \xi_{k+1}^- \end{array} \right]$$

where

$$\xi_{k+1}^+ = \gamma_{k+1}^{-1} (1 - v_{k+1}^T x^{(k)})$$

and

$$\xi_{k+1}^- = \gamma_{k+1}^{-1}(-1 - v_{k+1}^T x^{(k)})$$

and

$$v_j = (c_{1,j}^{(k)}, \dots, c_{k-1,j}^{(k)})^T$$

is the j^{th} column of C after H_1, H_2, \ldots, H_k are applied. Note that the last entry of v_j is not known until after we use column k to form H_k . The ρ_j can be accumulated through out the computation. For Weights $t_1, t_2, \ldots, t_n > 0$, we then examine

$$\zeta^{+} = |\xi_{k+1}^{+}| + \sum_{j=k+1}^{n} t_{j} \rho_{j}^{+}$$

and

$$\zeta^{-} = |\xi_{k+1}^{-}| + \sum_{j=k+1}^{n} t_{j} \rho_{j}^{-}$$

and choose

$$x^{(k+1)} = (x^{(k)}, \xi_{k+1}^+)^T$$

or

$$x^{(k+1)} = (x^{(k)}, \xi_{k+1}^-)^T$$

according to whether ζ^+ or ζ^- is larger. The choice of weights is heuristic. LINPACK chooses $t_j = u_{j,j}^{-1}$. However, we have not computed $u_{j,j}$ at this point. So we choose $t_j = \gamma_j^{-1}$.

We now explain how this can be used in column selection. Our algorithm performs the condition estimator on the most recently formed diagonal block C_{ii} until

- 1. it finds a zero diagonal
- 2. the estimate of $||C_{ii}^{-1}||$ exceeds ε^{-1} where ε is a predefined tolerance and is usually $O(\mu)$, μ being the machine precision.

In both cases, we restart the condition estimator with all $\rho_i = 0$ (implicitly) and then begin forming $C_{i+1,i+1}$. Of course, in case 2, we must find a dependent column in C_{ii} . To do this, we solve

$$C_{ii}h = e_k$$
.

Let ν be the index such that

$$|h_{\nu}|=\max_{1\leq j\leq k}|h_j|.$$

And we delete the column ν from C_{ii} and retriangularize the new matrix by a sequence of Given's rotations. The general idea of the algorithm is detailed below.

```
/* \epsilon \approx \mu is a tolerance factor.
c^{[l]} denotes the l^{th} row of C. */
done \leftarrow false
k \leftarrow 1
firstk \leftarrow 1
firstl \leftarrow 1
q_i \leftarrow i, i = 1, 2, \ldots n
\rho_i \leftarrow 0, i = 1, 2, \ldots n
\gamma_i \leftarrow ||c_i||^2, i = 1, 2, \dots n
\sigma \leftarrow 0
while not done do
\xi \leftarrow \gamma_k^{-1}(-sign(\rho_k) - \rho_k)
\hat{\sigma} \leftarrow max\{\sigma, |\xi|\}
if \hat{\sigma}^{-1} > \epsilon then
    construct an orthogonal transformation
    such that the current column is zeroed
    /* this column is good */
   l \leftarrow l + 1 and q_l \leftarrow k
   for j \in Nonz(c^{[l]})
          update the column norms \gamma_i
   \begin{array}{l} \xi^+ \leftarrow \gamma_k^{-1} (1 - \rho_k) \\ \xi_- \leftarrow \gamma_k^{-1} (-1 - \rho_k) \end{array}
    \zeta_{max}^+ \leftarrow |\xi^+|
    \zeta_{max}^- \leftarrow |\xi^-|
    for j \in Nonz(c^{[l]}) update the partial sums
          \rho_j^+ \leftarrow \rho_j + c_{l+1} \xi^+
          \rho_{j}^{-} \leftarrow \rho_{j} + c_{l+1}\xi^{-}
\zeta_{max}^{+} \leftarrow max\{\zeta_{max}^{+}, |\rho_{j}^{+}/\gamma_{j}|\}
          \zeta_{max}^- \leftarrow max\{\zeta_{max}^-, |\rho_j^-/\gamma_j|\}
    /* We just looked ahead of the affected columns
    and computed both possible values for
    the partial products */
    if \zeta_{max}^+ \geq \zeta_{max}^- then
       \sigma \leftarrow max\{\sigma, |\xi^+|\}
       \rho_j \leftarrow \rho_j^+, \quad j \in Nonz(c^{[l]})
       \sigma \leftarrow max\{\sigma, |\xi^-|\}

\rho_j \leftarrow \rho_j^-, \quad j \in Nonz(c^{[l]})

else the column is not good
    if \gamma_{max} \leq \epsilon then
       /* no more good columns and we are done */
       exit
    else
       Let C_{ii} be the submatrix of C
       with rows from firstl through l
       and columns from firstk through k.
```

```
Solve C_{ii}h = (0,0,\ldots,1)^T.

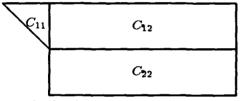
Let |h_{\nu}| = \max_i(|h_i|)

/* Break ties arbitrarily in choosing the maximum element */
Move the column \nu to the last column of C_{ii} and re-triangularize C_{ii}.

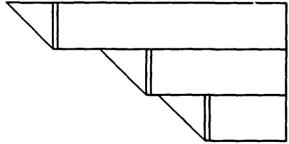
firstl \leftarrow 1
firstk \leftarrow 1
\rho_i \leftarrow 0 /* starting over a new block */
endif
endif
k \leftarrow k + 1
done \leftarrow (l \ge m_1 \text{ or } k \ge n)
endwhile
```

Figure 1: Factorization Algorithm with the Incremental Condition Estimator

As soon as a bad column is encountered, the matrix being factored will have the appearance as shown below.



Since a back-solve and a re-triangularization is done every time a bad column is encountered, we need a column which is a full vector. But fortunately, we can just reserve one vector (whose size is equal to the number of rows) to store the intermediate computations. The typical structure of the matrix after the factorization is shown below.



The final upper trapezoidal form of C may have the following form.

$$\left(\begin{array}{cccc} C_{11} & C_{12} & C_{13} & C_{14} \\ 0 & C_{22} & C_{23} & C_{24} \\ 0 & 0 & C_{33} & C_{34} \end{array}\right)$$

where C_{ii} , i = 1, 2, 3 have full row rank.

5. Issues in Parallel Implementation

The problem is to detect the rank of a large and sparse matrix C and obtain a QR factorization of that matrix. As was described in section 2, the general strategy is followed. We use SPARSPAK-B [8] for doing steps 1-3 as described in section 2. Form the storage structure provided for R by SPARSPAK, we generate the static structure required for the factorization. Since this work involves only the symbolic structure of C and is a well understood problem, we perform only the numerical factorization part on the parallel machine.

We consider issues in implementing this algorithm on a Hypercube architecture. In a rather straightforward way, the columns of the matrix C are wrapped around among the processors on the hypercube. For the sake of simplicity, we may assume that the processors form a ring, although for "broadcast" purposes, other connections of the hypercube are implicitly made use of. Each processor makes a decision as to include the next column in the factorization and sends a message to other processors along with the necessary transformations (if the column is included). The updating of the rest of the columns on the same processor is done only after sending the information to other processors.

There are a couple of obvious bottlenecks to this algorithm when implemented on a parallel machine. The "back-solve" process, when a bad column is found, involves accessing the partially formed upper triangular factor. This also causes the algorithm to come to a virtual pause, loosing some of the advantages of the asynchronous behavior of the algorithm. However, this happens only occasionally, so we can still expect some good speed-ups.

The "look-ahead" part of the algorithm, where it needs to find out which value of ρ is to be made permanent, is another bottleneck. There are a couple ways to get around this problem. We can make the look-ahead local to the columns held by that processor only. But then, we may be compromising on the quality of the estimate computed by the algorithm. The other alternative is to maintain a fixed number of possibilities(say z=4) of the values of ρ and in the steady state, by the time we each processor is ready to process a column y, it would have enough information to fix the value of ρ corresponding to column (y-z).

This algorithm was implemented on a Intel iPSC/2 Hypercube and the Table 1 summarizes the speed-ups that have been obtained. The test matrix with 4000 columns and approximately 30000 non-zero elements in the factored matrix, was generated randomly with

Table 1: Timing results on a random sparse matrix

no. of processors	time (secs)
2	15.20
4	10.78
8	7.76
16	5.54

Table 2: Results of our condition estimation tests

(k_2	n = 10	25	50
1	10	0.36/0.67	0.33/0.53	0.30/0.43
1	10^{3}	0.20/0.58	0.20/0.42	0.22/0.37
1	10 ⁶	0.11/0.48	0.12/0.36	0.10/0.27
ı	10 ⁹	0.12/0.51	0.12/0.33	0.09/0.26

a random sparsity structure. The speed-up obtained is by a factor of 1.4 for doubling the number of processors.

6. Robustness of the algorithm

To test the effectiveness of this estimator, we used this algorithm to estimate the condition number of a given matrix. As was suggested by Stewart [12], we generated random test matrices of dimension 10, 25 and 50 with a known condition number — the values being 1.0E1, 1.0E3, 1.0E6 and 1.0E9. For each of the possibilities, we generated two types of matrices — one where there is a sharp break in the singular value distribution and the other in which the singular values are exponentially distributed between 1 and the condition number.

The algorithm always estimated correctly (within 2 decimal digit accuracy), if there is a sharp break in the singular value distribution and hence the results in Table 2 only illustrate the case where there is a exponential distribution of singular values. For each dimension n, 50 test matrices were generated. k_2 is the actual condition number of the test matrix. The numbers quoted in each entry represent the minimum / average value of the ratio of the estimated condition number to the actual value. The results are rounded to two significant digits, so a ratio of 1.0 implies that the estimate had at least 2 correct digits.

Comparative results are included in Table 3 for LINPACK (taken from Higham [11]) and in Table 4 for Bischof's estimator (taken from Bischof [1]).

Table 3: Results of LINPACK condition estimation tests

k_2	n = 10	25	50
10	0.29/0.46	0.24/0.30	0.17/0.23
10 ³	0.29/0.56	0.20/0.33	0.19/0.26
106	0.46/0.76	0.20/0.46	0.22/0.35
10 ⁹	0.68/0.86	0.24/0.55	0.23/0.40

Table 4: Results of Bischof's condition estimation tests

k_2	n = 10	25	50
10	0.56/0.77	0.59/0.71	0.63/0.71
10 ³	0.33/0.53	0.40/0.50	0.31/0.45
10 ⁶	0.12/0.53	0.16/0.38	0.24/0.36
10 ⁹	0.16/0.45	0.17/0.33	0.19/0.31

7. Conclusions

We proposed an incremental condition estimation algorithm that is well suited for parallel sparse matrix factorizations. Empirical results provided suggest that the estimator is robust enough. Good speedups have been demonstrated on randomly generated test problems.

8. Acknowledgement

The authors would like to thank the Oak Ridge National Laboratory for providing the access to the usage of the Intel Hypercube.

References

- [1] C. H. Bischof. QR Factorization Algorithms for Coarse-Grained Distributed Systems. PhD thesis, Cornell University, August 1988.
- [2] C. H. Bischof, John G. Lewis, and Daniel J. Pierce. Incremental condition estimation for general matrices. Technical Report MCS-P106-0989, Mathematics and Computer sciences division, Argonne National Laboratory, Argonne, IL, September 1989.
- [3] Christan H. Bischof. Incremental condition estimation. SIAM Journal on Matrix Analysis and Applications, 11(2):312-322, April 1990.
- [4] A. K. Cline, A. R. Conn, and Charles F. Van Loan. Generalizing the LINPACK condition estimator, volume 909 of Lecture Notes in Mathematics. Springer-Verlag, Berlin, 1982.

- [5] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. LINPACK User's Guide. SIAM Publications, Philadelphia, 1979.
- [6] J. A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. Linear Algebra and Its Applications, 34:69-83, 1980.
- [7] J. A. George and J. W. H. Liu. Computer Solution of Large Sparse Positive Definite Systems. Prentice Hall, Englewood Cliffs, N. J., 1980.
- [8] J. A. George and Esmond Ng. SPARSPAK: Waterloo sparse matrix package user's guide for SPARSPAK-B. Technical Report CS-84-37, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, November 1984.
- [9] Gene H. Golub and Charles F. Van Loan. Matrix Computations. The Johns Hopkins Press, Baltimore, 1983.
- [10] M. T. Heath. Some extensions of an algorithm for sparse linear least squares problems. SIAM Journal on Scientific and Statistical Computing, 3:223-237, 1982.
- [11] Nicholas J. Higham. A survey of condition number estimation for triangular matrices. SIAM Review, 29(4):575-596, 1987.
- [12] G. W. Stewart. The efficient generation of random orthogonal matrices with an application to condition estimators. SIAM Journal on Numerical Analysis, 17:403-9, 1980.
- [13] Charles F. Van Loan. On estimating the condition of eigenvalues and eigenvectors. *Linear Algebra and Its Applications*, 88/89:715-732, 1987.

LU Factorization of Sparse, Unsymmetric Jacobian Matrices on Multicomputers: Experience, Strategies, Performance

STUDENT PAPER

Anthony Skjellum Alvin P. Leung

Advisor: Manfred Morari

California Institute of Technology
Chemical Engineering; mail code 210-41
Pasadena, California 91125
e-mail: tony@perseus.ccsf.caltech.edu

Abstract

Efficient sparse linear algebra cannot be achieved as a straightforward extension of the dense case, even for concurrent implementations. This paper details a new, general-purpose unsymmetric sparse LU factorization code built on the philosophy of Harwell's MA28, with variations. We apply this code in the framework of Jacobian-matrix factorizations, arising from Newton iterations in the solution of nonlinear systems of equations. Serious attention has been paid to the data-structure requirements, complexity issues and communication features of the algorithm. Key results include reduced communication pivoting for both the "analyze" A-mode and repeated B-mode factorizations, and effective general-purpose data distributions useful incrementally to trade-off process-column load balance in factorization against triangular solve performance. Future planned efforts are cited in conclusion.

Introduction

The topic of this paper is the implementation and concurrent performance of sparse, unsymmetric LU factorization for medium-grain multicomputers. Our target hardware is distributed-memory, message-passing concurrent computers such as the Symult s2010 and Intel iPSC/2 systems. For both of these systems, efficient cut-through wormhole routing technology provides pair-wise communication performance essentially independent of the spatial location of the computers in the ensemble [2]. The Symult s2010 is a two-dimensional, mesh-connected concurrent computer; all examples in this paper were run on this variety of hardware. Message-passing performance, portability and

sive throughout scientific and engineering computation. The need for high-quality, high-performance linear algebra algorithms (and libraries) for multicomputer systems therefore requires no attempt at justification. The motivation for the work described here has a specific origin, however. Our main higher-level research goal is the concurrent dynamic simulation of systems modelled by ordinary differential and algebraic equations; specifically, dynamic flowsheet simulation of chemical plants (e.g., coupled distillation columns) [8]. Efficient sequential integration algorithms solve staticized nonlinear equations at each time point via modified Newton iteration (cf., [3], Chapter 5). Consequently, a sequence of structurally identical linear systems must be solved; the matrices are finite-difference approximations to Jacobians of

the staticized system of ordinary differential-algebraic equations. These Jacobians are large, sparse and un-

symmetric for our application area. In general, they possess both band and significant off-band structure. Generic structures are depicted in Figure 0. This

work should also bear relevance to electric power net-

work/grid dynamic simulation where sparse, unsym-

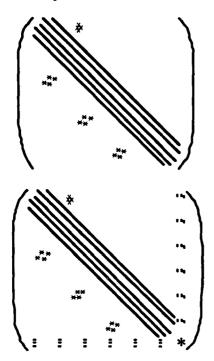
metric Jacobians also arise, and also elsewhere.

related issues relevant to this work are detailed in [7]. Questions of linear-algebra performance are perva-

Design Overview

We solve the problem Ax = b where A is large, and includes many zero entries. We assume that A is unsymmetric both in sparsity pattern and in numerical values. In general, the matrix A will be computed in a distributed fashion, so we will inherit a distribution of the coefficients of A (cf., Figures 2., 3.). Follow-

Figure 0. Example Jacobian Matrix Structures.



In chemical-engineering process flowsheets, Jacobians with main band structure, and lower-triangular structure (feedforwards), upper-triangular structure (feedbacks), and borders (global or artificially restructured feedforwards and/or feedbacks) are common.

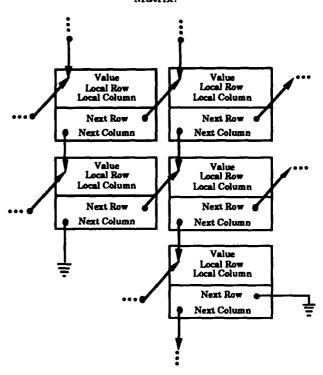
ing the style of Harwell's MA28 code for unsymmetric sparse matrices, we use a two-phase approach to this solution. There is a first LU factorization called Amode or "analyze," which builds data structures dynamically, and uses a user-defined pivoting function. The repeated B-mode factorization uses the existing data structures statically to factor a new, similarly structured matrix, with the previous pivoting pattern. B-mode monitors stability with a simple growth factor estimate. In practice, A-mode is repeated whenever instability is detected. The two key contributions of this sparse concurrent solver are: reduced communication pivoting, and new data distributions for better overall performance.

Following Van de Velde [11], we consider the LU factorization of a real matrix A, $A \in \Re^{N \times N}$. It is well known (e.g., [6], pp. 117-118), that for any such matrix A, an LU factorization of the form

$$P_R A P_C^T = \hat{L} \hat{U}$$

exists, where P_R , P_C are square, (orthogonal) permutation matrices, and \hat{L} , \hat{U} are the unit lower-triangular,

Figure 1. Linked-list Entry Structure of Sparse Matrix.



A single entry consists of a double-precision value (8 bytes), the local row (i) and column (j) index (2 bytes each), a "Next Column Pointer" indicating the next current column entry (fixed j), and a "Next Row Pointer" indicating the next current row entry (fixed i), at 4 bytes each. Total: 24 bytes per entry.

and upper-triangular factors, respectively. Whereas the pivot sequence is stored (two N-length integer vectors), the permutation matrices are not stored or computed with explicitly. Rearranging, based on the orthogonality of the permutation matrices, $A = P_R^T \hat{L} \hat{U} P_C$. We factor A with implicit pivoting (no rows or columns are exchanged explicitly as a result of pivoting). Therefore, we do not store \hat{L}, \hat{U} directly, but instead: $L = P_R^T \hat{L} P_C$, $U = P_R^T \hat{U} P_C$. Consequently, $\hat{L} = P_R L P_C^T$, $\hat{U} = P_R U P_C^T$, and $A = L(P_C^T P_R)U$. The "unravelling" of the permutation matrices is accomplished readily (without implication of additional interprocess communication) during the triangular solves.

For the sparse case, performance is more difficult to quantify than for the dense case, but, for example, banded matrices with bandwidth β can be factored with $O(\beta^2 N)$ work; we expect sub-cubic complexity in N for reasonably sparse matrices, and strive for sub-quadratic complexity, for very sparse matrices. The

triangular solves can be accomplished in work proportional to the number of entries in the respective triangular matrix L or U. The pivoting strategy is treated as a parameter of the algorithm and is not pre-determined. We can consequently treat the pivoting function as an application-dependent function, and sometimes tailor it to special problem structures (cf., Section 7 of [9]) for higher performance. As for all sparse solvers, we also seek sub-quadratic memory requirements in N, attained by storing matrix entries in linked-list fashion, as illustrated in Figure 1.

For further discussion of LU factorizations and sparse matrices, see [6,4].

Reduced-Communication Pivoting

At each stage of the concurrent LU factorization, the pivot element is chosen by the user-defined pivot function. Then, the pivot row (new row of U) must be broadcast, and pivot column (new column of L) must be computed and broadcast on the logical process grid (cf., Figure 2.), vertically and horizontally, respectively. Note that these are interchangeable operations. We use this degree-of-freedom to reduce the communication complexity of particular pivoting strategies, while impacting the effort of the LU factorization itself negligibly.

We define two "correctness modes" of pivoting functions. In the first correctness mode "first row fanout." the exit conditions for the pivot function are: all processes must know \hat{p} (the pivot process row), the pivot process row must know \hat{q} (the pivot process column) as well as \hat{i} , the \hat{p} -local matrix row of the pivot, and the pivot process must know in addition the pivot value and \hat{q} -local matrix column \hat{j} of the pivot. Partial column pivoting and preset pivoting can be setup to satisfy these correctness conditions as follows. For partial column pivoting, the kth row is eliminated at the kth step of the factorization. From this fact, each process can derive the process row \hat{p} and \hat{p} -local matrix row \hat{i} using the row data distribution function. Having identified themselves, the pivot-row processes can look for the largest element in local matrix row î and choose the pivot element globally among themselves via a combine. At completion this places \hat{q} , \hat{j} and the pivot value in the entire pivot process row. This completes the requirements for the "first row fanout" correctness mode. For preset pivoting, the kth elimination row and column are both stored as \hat{p} , \hat{i} , \hat{q} , \hat{j} , and each process knows these values without communication. 1 Furthermore, the pivot process looks up the pivot value. Hence, preset pivoting satisfies the requirements of this correctness mode also.

For "first row fanout," the universal knowledge of \hat{p} and knowledge of the pivot matrix row $\hat{\imath}$ by the pivot process row, allows the vertical broadcast of this row (new row of U). In addition, we broadcast \hat{q} , \hat{j} and the pivot value simultaneously. This extends the correct value of \hat{q} to all processes, as well as \hat{j} and the pivot value to the pivot process column. Hence, the multiplier (L) column may be correctly computed and broadcast. Along with the multiplier column broadcast, we include the pivot value. After this broadcast, all processes have the correct indices $\hat{p}, \hat{\imath}, \hat{q}, \hat{\jmath}$ and the pivot value. This provides all that's required to complete the current elimination step.

For the second correctness mode "first column fanout," the exit conditions for the pivot function are: all processes must know \hat{q} , the entire pivot process column must know \hat{j} , the pivot value, and \hat{p} . The pivot process in addition knows \hat{i} . Partial row pivoting can be setup to satisfy these correctness conditions. The arguments are analogous to partial column pivoting and are given in [8].

For "first column fanout," the entire pivot process column knows the pivot value, and local column of the pivot. Hence, the multiplier column may be computed by dividing the pivot matrix column by the pivot value. This column of L may then be broadcast horizontally, including the pivot value, \hat{p} and \hat{i} as additional information. After this step, the entire ensemble has the correct pivot value, and \hat{p} ; in addition, the pivot process row has the correct \hat{i} . Hence, the pivot matrix row may be identified and broadcast. This second broadcast completes the needed information in each process for effecting the kth elimination step.

Hence, when using partial row or partial column pivoting, only local combines of the pivot process column (respectively row) are needed. The other processes don't participate in the combine, as they must without this methodology. Preset pivoting implies no pivoting communication, except very occasionally (e.g., 1 in 5000 times) as noted in [8] to remove memory unscalabilities. This pivoting approach is a direct savings, gained at a negligible additional broadcast overhead. See also [8].

New Data Distributions

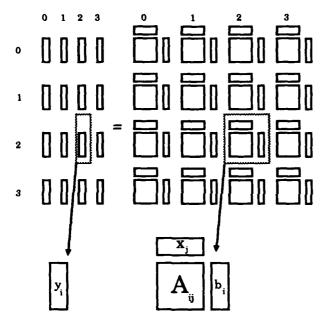
We introduce new closed-form O(1)-time, O(1)-memory data distributions useful for sparse matrix factorizations and the problems that generate such matrices. We quantify evaluation costs in Table 0. Every concurrent data structure is associated with a logi-

¹Memory unscalabilities can be removed very cheaply; see [8].

Table 0. Data-Distribution Function and Inverse Costs				
Distribution:	$\mu(I,P,M)$	$\mu^{-1}(p,i,P,M)$		
One-Parameter (ζ)	$5.5554 \times 10^{1} \pm 5 \times 10^{-3}$	$4.0024 \times 10^{1} \pm 7 \times 10^{-3}$		
Two-Parameter (ξ)	$6.1710 \times 10^1 \pm 1 \times 10^{-2}$	$4.2370 \times 10^{1} \pm 8 \times 10^{-3}$		
Block-Linear (λ)	$5.4254 \times 10^{1} \pm 7 \times 10^{-3}$	$3.5404 \times 10^{1} \pm 5 \times 10^{-3}$		

For the data distributions and inverses described here, evaluation time in μ s is quoted for the Symult s2010 multicomputer. Cardinality function calls are inexpensive, and fall within lower-order work anyway – their timing is hence omitted. The cheapest distribution function (scatter) costs $\approx 15\mu$ s by way of comparison.

Figure 2. Process Grid Data Distribution of Ax = b.



Representation of a concurrent matrix, and distributedreplicated concurrent vectors on a 4x4 logical process grid. The solution of Ax = b first appears in x, a columndistributed vector, and then is normally "transposed" via a global *combine* to the row-distributed vector y.

cal process grid at creation (cf., Figure 2. and [7,8]). Vectors are either row- or column-distributed within a two-dimensional process grid. Row-distributed vectors are replicated in each process column, and distributed in the process rows. Conversely, column-distributed vectors are replicated in each process row, and distributed in the process columns. Matrices are distributed both in rows and columns, so that a single process owns a subset of matrix rows and columns. This partitioning follows the ideas proposed by Fox et al. [5] and others. Within the process grid, coefficients

of vectors and matrices are distributed according to one of several data distributions. Data distributions are chosen to compromise between load-balancing requirements and constraints on where information can be calculated in the ensemble.

Definition 1 (Data-Distribution Function)

A data-distribution function μ maps three integers $\mu(I,P,M) \mapsto (p,i)$ where $I, 0 \leq I < M$, is the global name of a coefficient, P is the number of processes among which all coefficients are to be partitioned, and M is the total number of coefficients. The pair (p,i) represents the process $p(0 \leq p < P)$ and local (process-p) name i of the coefficient $(0 \leq i < \mu^{l}(p,P,M))$. The inverse distribution function $\mu^{-1}(p,i,P,M) \mapsto I$ transforms the local name i back to the global coefficient name I.

The formal requirements for a data distribution function are as follows. Let \mathcal{I}^p be the set of global coefficient names associated with process p, $0 \le p < P$, defined implicitly by a data distribution function $\mu(\bullet, P, M)$. The following set properties must hold:

$$\mathcal{I}^{p_1} \cap \mathcal{I}^{p_2} = \emptyset, \ \forall \ p_1 \neq p_2, \quad 0 \leq p_1, p_2 < P$$

$$\bigcup_{p=0}^{P-1} \mathcal{I}^p = \{0, \dots, M-1\} \equiv \mathcal{I}_M$$

The cardinality of the set \mathcal{I}^p , is given by $\mu^{\dagger}(p, P, M)$.

The linear and scatter data-distribution functions are most often defined. We generalize these functions (by blocking and scattering parameters) to incorporate practically important degrees of freedom. These generalized distribution functions yield optimal static load balance as do the unmodified functions described in [11] for unit block size, but differ in coefficient placement. This distinction is technical, but necessary for efficient implementations.

Definition 2 (Generalized Block-Linear)

The definitions for the generalized block-linear distribution function, inverse, and cardinality function are:

$$\begin{array}{rcl} \lambda_B(I,P,M) & \mapsto & (p,i), \\ p & \equiv & P-1- \\ & & \max\left(\left\lfloor\frac{I_B^{\rm rev}}{l+1}\right\rfloor, \left\lfloor\frac{I_B^{\rm rev}-r}{l}\right\rfloor\right), \\ i & \equiv & I-B\left(pl+\Theta^1\left(p-(P-r)\right)\right), \end{array}$$

while

$$\lambda_B^{-1}(p, i, P, M) \equiv i + B \left((pl + \Theta^1 (p - (P - r))) \right),$$

$$\lambda_B^{1}(p, P, M) \equiv B \left(\left\lfloor \frac{b + p}{P} \right\rfloor - \theta \right) +$$

$$(M \mod B)\theta,$$

where B denotes the coefficient block size,

$$b = \begin{cases} \frac{M}{B} & \text{if } M \mod B = 0 \\ \left\lfloor \frac{M}{B} \right\rfloor + 1 & \text{otherwise,} \end{cases}$$

$$I_B = \left\lfloor \frac{I}{B} \right\rfloor, \quad I_B^{\text{rev}} = b - 1 - I_B,$$

$$l = \left\lfloor \frac{b}{P} \right\rfloor, \quad r = b \mod P,$$

$$\Theta^k(t) \equiv \begin{cases} 0 & t \le 0 \\ t^k & t > 0, \ k > 0 \\ 1 & t > 0, \ k = 0 \end{cases}$$

$$\theta = \left\lfloor \frac{p+1}{P} \right\rfloor \Theta^0(M \mod B)$$

and where $b \geq P$.

For B=1, a load-balance-equivalent variant of the common linear data-distribution function is recovered. The general block-linear distribution function divides coefficients among the P processes $p=0,\ldots,P-1$ so that each \mathcal{I}^p is a set of coefficients with contiguous global names, while optimally load-balancing the b blocks among the P sets. Coefficient boundaries between processes are on multiples of B. The maximum possible coefficient imbalance between processes is B. If $B \mod P \neq 0$, the last block in process P-1 will be foreshortened.

Definition 3 (Parametric Functions)

To allow greater freedom in the distribution of coefficients among processes, we define a new, two-parameter distribution function family, ξ . The B blocking parameter (just introduced in the block-linear

function) is mainly suited to the clustering of coefficients that must not be separated by an interprocess boundary (again, see [8] for a definition of general block-scatter, σ). Increasing B worsens the static load balance. Adding a second scaling parameter S (of no impact on the static load balance) allows the distribution to scatter coefficients to a greater or lesser degree, directly as a function of this one parameter. The two-parameter distribution function, inverse and cardinality function are defined below. The one-parameter distribution function family, ζ , occurs as the special case B=1, also as noted below:

$$\xi_{B,S}(I,P,M) \mapsto (p,i) \equiv \begin{cases} (p_0,i_0) & \Lambda_0 \geq l_S \\ (p_1,i_1) & \Lambda_0 < l_S \end{cases}$$

where

$$l_S \equiv \left\lfloor \frac{l}{S} \right\rfloor, \quad \Lambda_0 \equiv \left\lfloor \frac{i_0}{BS} \right\rfloor,$$
 $(p_0, i_0) \leftarrow \lambda_B(I, P, M),$
 $I_{BS} = p_0 l_S + \Lambda_0,$
 $p_1 \equiv I_{BS} \mod P,$
 $i_1 \equiv BS \left\lfloor \frac{I_{BS}}{P} \right\rfloor + (i_0 \mod BS),$

with

$$\zeta_{B,S}(I,P,M) \equiv \xi_{1,S}(I,P,M),
\xi_{B,S}^{\dagger}(p,P,M) \equiv \zeta_{S}^{\dagger}(p,P,M)
\equiv \lambda_{B}^{\dagger}(p,P,M),$$

and where r, b, etc. are as defined above. The inverse distribution function ξ^{-1} is defined as follows:

$$\xi_{B,S}^{-1}(p,i,P,M) \mapsto I = \lambda_B^{-1}(p^*,i^*,P,M),
(p^*,i^*) \equiv \begin{cases} (p,i) & \Lambda \ge l_S \\ (p_2,i_2) & \Lambda < l_S \end{cases},
\Lambda \equiv \begin{bmatrix} \frac{i}{BS} \end{bmatrix}, I_{BS}^* = p + \Lambda P,
p_2 \equiv \begin{bmatrix} \frac{I_{BS}^*}{l_S} \end{bmatrix},
i_2 \equiv BS(I_{BS}^* \mod l_S) + (i \mod BS),$$

with

$$\zeta_S^{-1}(p, i, P, M) \equiv \xi_{1,S}^{-1}(p, i, P, M).$$

For S=1, a block-scatter distribution results, while for $S \geq S_{crit} \equiv l_S + 1$, the generalized block-linear distribution function is recovered. See also [8].

Definition 4 (Data Distributions)

Given a data-distribution function family $(\mu, \mu^{-1}, \mu^{\parallel})$ $((\nu, \nu^{-1}, \nu^{\parallel}))$, a process list of P(Q), M(N) as the number of coefficients, and a row (respectively, column) orientation, a row (column) data distribution \mathcal{G}^{row} (\mathcal{G}^{col}) is defined as:

$$\mathcal{G}^{row} \equiv \left\{ (\mu, \mu^{-1}, \mu^{\parallel}); P, M \right\},\,$$

respectively,

$$\mathcal{G}^{col} \equiv \left\{ (\nu, \nu^{-1}, \nu^{\parallel}); Q, N \right\}.$$

A two-dimensional data distribution may be identified as consisting of a row and column distribution defined over a two-dimensional process grid of $P \times Q$ processes, as $\mathcal{G} \equiv (\mathcal{G}^{row}, \mathcal{G}^{col})$.

Further discussion and detailed comparisons on datadistribution functions are offered in [8]. Figure 3. illustrates the effects of linear and scatter data-distribution functions on a small rectangular array of coefficients.

Performance vs. Scattering

Consider a fixed logical process grid of R processes. with PxQ = R. For the sake of argument, assume partial row pivoting during LU factorization for the retention of numerical stability. Then, for the LU factorization, it is well known that a scatter distribution is "good" for the matrix rows, and optimal were there no off-diagonal pivots chosen. Furthermore, the optimal column distribution is also scatter, because columns are chosen in order for partial row pivoting. Compatibly, a scatter distribution of matrix rows is also "good" for the triangular solves. However, for triangular solves, the best column distribution is linear, because this implies less intercolumn communication, as we detail below. In short, the optimal configurations conflict, and because explicit redistribution is expensive, a static compromise must be chosen. We address this need to compromise through the one-parameter distribution function (described in the previous section, offering a variable degree of scattering via the Sparameter. To first order, changing S does not affect the cost of computing the Jacobian (assuming columnwise finite-difference computation), because each process column works independently.

It's important to note that triangular solves derive no benefit from Q>1. The standard column-oriented solve keep one process column active at any given time. For any column distribution, the updated right-hand-side vectors are retransmitted W times (process column-to-process column) during the triangular solve

- whenever the active process column changes. There are at least $W_{min} \equiv Q-1$ such transmissions (linear distribution), and at most $W_{max} \equiv N-1$ transmissions (scatter distribution). The complexity of this retransmission is O(WN/P), representing quadratic work in N for $W \sim N$.

Calculation complexity for a sparse triangular solve is proportional to the number of elements in the triangular matrix, with a low leading coefficient. Often, there are $O(N^{1.x})$ with x < 1 elements in the triangular matrices, including fill. This operation is then $O(N^{1.x}/P)$, which is less than quadratic in N. Consequently, for large W, the retransmission step is likely of greater cost than the original calculation. This retransmission effect constrains the amount of scattering and size of Q in order to have any chance of concurrent speedup in the triangular solves.

Using the one-parameter distribution with $S \geq 1$ implies that $W \approx N/S$, so that the retransmission complexity is $O(N^2/SP)$. Consequently, we can bound the amount of retransmission work by picking S sufficiently large. Clearly, $S = S_{crit}$ is a hard upper bound, because we reach the linear distribution limit at that value of the parameter. We suggest picking $S \approx 10$ as a first guess, and $S \sim \sqrt{N}$, more optimistically. The former choice basically reduces retransmission effort by an order of magnitude. Both examples in the following section illustrate the effectiveness of choosing S by these heuristics.

The two-parameter ξ distribution can be used on the matrix rows to tradeoff load balance in the factorizations and triangular solves against the amount of (communication) effort needed to compute the Jacobian. In particular, a greater degree of scattering can dramatically increase the time required for a Jacobian computation (depending heavily on the underlying equation structure and problem), but importantly reduce load imbalance during the linear algebra steps. The communication overhead caused by multiple process rows suggests shifting toward smaller P and larger Q (a squatter grid), in which case greater concurrency is attained in the Jacobian computation, and the additional communication previously induced is then somewhat mitigated. The one-parameter distribution used on the matrix columns then proves effective in controlling the cost of the triangular solves by choosing the minimally allowable amount of column scattering.

Let's make explicit the performance objectives we consider when tuning S, and, more generally, when tuning the grid shape PxQ = R. In the modified Newton iteration, for instance, a Jacobian factorization is reused until convergence slows unacceptably. An "LU Factor-

Figure 3. Example of Process-Grid Data Distribution

$$\begin{pmatrix} A^{0,0} & A^{0,1} & A^{0,2} & A^{0,3} \\ A^{1,0} & A^{1,1} & A^{1,2} & A^{1,3} \\ A^{2,0} & A^{2,1} & A^{2,2} & A^{2,3} \\ A^{3,0} & A^{3,1} & A^{3,2} & A^{3,3} \end{pmatrix}_{\mathcal{G}} = \begin{pmatrix} a_{0,1} & a_{0,5} & a_{0,2} & a_{0,6} & a_{0,3} & a_{0,7} & a_{0,0} & a_{0,4} & a_{0,8} \\ a_{1,1} & a_{1,5} & a_{1,2} & a_{1,6} & a_{1,3} & a_{1,7} & a_{1,0} & a_{1,4} & a_{1,8} \\ a_{2,1} & a_{2,5} & a_{2,2} & a_{2,6} & a_{2,3} & a_{2,7} & a_{2,0} & a_{2,4} & a_{2,8} \\ a_{3,1} & a_{3,5} & a_{3,2} & a_{3,6} & a_{3,3} & a_{3,7} & a_{3,0} & a_{3,4} & a_{3,8} \\ a_{4,1} & a_{4,5} & a_{4,2} & a_{4,6} & a_{4,3} & a_{4,7} & a_{4,0} & a_{4,4} & a_{4,8} \\ a_{5,1} & a_{5,5} & a_{5,2} & a_{5,6} & a_{5,3} & a_{5,7} & a_{5,0} & a_{5,4} & a_{5,8} \\ a_{6,1} & a_{6,5} & a_{6,2} & a_{6,6} & a_{6,3} & a_{6,7} & a_{6,0} & a_{6,4} & a_{6,8} \\ a_{7,1} & a_{7,5} & a_{7,2} & a_{7,6} & a_{7,3} & a_{7,7} & a_{7,0} & a_{7,4} & a_{7,8} \\ a_{8,1} & a_{8,5} & a_{8,2} & a_{8,6} & a_{8,3} & a_{8,7} & a_{8,0} & a_{8,4} & a_{8,8} \\ a_{9,1} & a_{9,5} & a_{9,2} & a_{9,6} & a_{9,3} & a_{9,7} & a_{9,0} & a_{9,4} & a_{9,8} \\ a_{10,1} & a_{10,5} & a_{10,2} & a_{10,6} & a_{10,3} & a_{10,7} & a_{10,0} & a_{10,4} & a_{10,8} \end{pmatrix}$$

An 11×9 array with block-linear rows (B = 2) and scattered columns on a 4×4 logical process grid. Local arrays are denoted at left by $A^{p,q}$ where (p,q) is the grid position of the process on $\mathcal{G} \equiv \left(\left\{(\lambda_2, \lambda_2^{-1}, \lambda_2^{\dagger}); P = 4, M = 11\right\}, \left\{(\sigma_1, \sigma_1^{-1}, \sigma_1^{\dagger}); Q = 4, N = 9\right\}\right)$. Subscripts (i.e., $a_{I,J}$) are the global (I,J) indices.

ization + Backsolve" step is followed by η "Forward + Backsolves," with $\eta \sim O(1)$ typically (and varying dynamically throughout the calculation). Assuming an averaged η , say η^* (perhaps as large as five [3]), then our first-level performance goal is a heuristic minimization of

$$T_{LU} + (\eta^* + 1)T_{Back} + \eta^*T_{Forward}$$

over S for fixed P, Q. $\eta^* > 1$ more heavily weights the reduction of triangular solve costs vs. B-mode factorization than we might at first have assumed, placing a greater potential gain on the one-parameter distribution for higher overall performance. We generally want heuristically to optimize

$$T_{Jac} + T_{LU} + (\eta^* + 1)T_{Back} + \eta^*T_{Forward}$$

over S, P, Q, R. Then, the possibility of fine-tuning row and column distributions is important, as is the use of non-power-of-two grid shapes.

Performance

Order 13040 Example

We consider an order 13040 banded matrix with a bandwidth of 326 under partial row pivoting. For this example, we have compiled timing results for a 16x12 process grid with random matrices (entries have range 0-10,000) using different values of S on the column distribution (see Table 1). We indicate timing for Amode, B-mode, Backsolves and Forward- and Backsolves together ("Solve" heading). For this example,

S=30 saves 76% of the triangular solve cost compared to S=1, or approximately 186 seconds, roughly 6 seconds above the linear optimal. Simultaneously, we incur about 17 seconds additional cost in B-mode, while saving about 93 seconds in the Backsolve. Assuming $\eta^*=1$ ($\eta^*=0$), in the first above-mentioned objective function, we save about 262 (respectively, 76) seconds. Based on this example, and other experience, we conclude that this is a successful practical technique for improving overall sparse linear algebra performance. The following example further bolsters this conclusion.

Order 2500 Example

Now, we turn to a timing example of an order 2500 sparse, random matrix. The matrix has a random diagonal, plus two-percent random fill of the off-diagonals; entries have a dynamic range of 0-10,000. Normally, data is averaged over random matrices for each grid shape (as noted), and over four repetitive runs for each random matrix. Partial row pivoting was used exclusively. Table 2. compiles timings for various grid shapes of row-scatter/column-scatter, and row-scatter / column-(S=10) distributions, for as few as nine nodes and as many as 128. Memory limitations set the lower bound on the number of nodes.

This example demonstrates that speedups are possible for this reasonably small sparse example with this general-purpose solver, and that the one-parameter distribution is key to achieving overall better performance even for this random, essentially unstructured example. Without the one-parameter distribution, triangular solver performance is poor, except in grid con-

	Table 1. Order 13040 Band Matrix Performance					
Distribution:		(time in seconds)			G-1	
Row	Column	A-Mode	B-Mode	Back-Solve	Solve	
Scatter	S=1	1.140×10^3	1.603×10^2	1.196×10^2	2.426×10^2	
	S=10	1.148×10^{3}	1.696×10^2	3.294×10^{1}	6.912×10^{1}	
	S=25	1.091×10^3	1.670×10^2	2.713×10^{1}	5.752×10^{1}	
	S=30	1.095×10^3	1.769×10^2	2.653×10^{1}	5.631×10^{1}	
	S=40	1.116×10^3	2.157×10^2	2.573×10^{1}	5.472×10^{1}	
	S=50	1.127×10^3	2.157×10^2	2.764×10^{1}	5.743×10^{1}	
	S=100	1.279×10^3	4.764×10^2	2.520×10^{1}	5.367×10^{1}	
	Linear	2.247×10^3	1.161×10^3	2.333×10^{1}	4.993×10^{1}	

The above timing data, for the 16x12 grid configuration with scattered rows, indicates the importance of the one-parameter distribution with S > 1 for balancing factorization cost vs. triangular-solve cost. The random matrices, of order 13040, have an upper bandwidth of 164 and a lower bandwidth of 162. "Best" performance occurs in the range $S \approx 25...40$.

figurations where the factorization is itself degraded (e.g., 2x16). Furthermore, the choice of S=10 is universally reasonable for the Q>1 grid shapes illustrated here, so the distribution proves easy to tune for this type of matrix. We are able to maintain an almost constant speed for the triangular solves while increasing speed for both the A-mode and B-mode factorizations. We presume, based on experience, that triangular solve times are comparable to the sequential solution times—further study is needed in this area to see if and how performance can be improved. The consistent A-mode to B-mode ratio of approximately two is attributed primarily to reduced communication costs in B-mode, realized through the elimination of essentially all combine operations in B-mode.

While triangular-solve performance exemplifies sequentialism in the algorithm, it should be noted that we do achieve significant overall performance improvements between 9 nodes and 72 (12x6 grid) nodes, and that the repeatedly used B-mode factorization remains dominant compared to the triangular solves even for 128 nodes. Consequently, efforts aimed further to increase performance of the B-mode factorization (at the expense of additional A-mode work) are interesting to consider. For the factorizations, we also expect that we are achieving non-trivial speedups relative to one node, but we are unable to quantify this at present because of the memory limitations alluded to above.

Future Work, Conclusions

There are several classes of future work to be considered. First, we need to take the A-mode "analyze" phase to its logical completion, by including pivot-order sorting of the L/U pointer structures to improve performance for systems that should demonstrate sub-quadratic sequential complexity. This will require minor modifications to B-mode (that already takes advantage of column-traversing elimination), to reduce testing for inactive rows as the elimination progresses. We already realize optimal computation work in the triangular solves, and we mitigate the effect of Q > 1 quadratic communication work using the one-parameter distribution.

Second, we need to exploit "timelike" concurrency in linear algebra – multiple pivots. This has been addressed by Alaghband for shared-memory implementations of MA28 with O(N)-complexity heuristics [1]. These efforts must be reconsidered in the multicomputer setting and effective variations must be devised. This approach should prove an important source of additional speedup for many chemical engineering applications, because of the tendency towards extreme sparsity, with mainly band and/or block-diagonal structure.

Third, we could exploit new communication strategies and data redistribution. Within a process grid, we could incrementally redistribute L/U by utilizing the inherent broadcasts of L columns and U rows to improve load balance in the triangular solves at

	Table 2. Order 2500 Matrix Performance						
	Distribution: (time in seconds)						
Shape	Row	Column	A-Mode	B-Mode	Back-Solve	Solve	Avgs
3x3	Scatter	Scatter	3.567×10^2	1.783×10^2	1.997×10^{1}	4.115×10^{1}	1
3x4		Scatter	3.101×10^2	1.303×10^2	2.149×10^{1}	4.452×10^{1}	1
4x3		Scatter	2.778×10^2	1.526×10^2	1.728×10^{1}	3.537×10^{1}	1
2x16		Scatter	4.500×10^2	3.350×10^2	3.175×10^{0}	1.101×10^{1}	1
12x1		Scatter	2.636×10^{2}	1.206×10^{2}	4.0188×10^{0}	8.340×10^{0}	3
16x1		Scatter	2.085×10^2	1.000×10^2	4.856×10^{0}	9.8744×10^{0}	3
8x2		Scatter	2.013×10^2	9.41×10^{1}	1.127×10^{1}	2.295×10^{1}	3
		S = 10	1.997×10^2	9.63×10^{1}	4.508×10^{0}	9.399×10^{0}	3
4x4		Scatter	2.371×10^2	1.056×10^2	1.225×10^{1}	3.549×10^{1}	3
		S = 10	2.329×10^2	1.104×10^2	4.192×10^{0}	9.406×10^{0}	3
4x6		Scatter	1.456×10^2	7.72×10^{1}	1.723×10^{1}	3.528×10^{1}	3
		S=10	1.684×10^{2}	8.85×10^{1}	4.206×10^{0}	9.303×10^{0}	3
12x2		Scatter	1.490×10^{2}	6.95×10^{1}	9.08×10^{0}	1.851×10^{1}	3
		S = 10	1.425×10^2	6.54×10^{1}	4.557×10^{0}	9.439×10^{0}	3
12x3		Scatter	1.0429×10^{2}	5.39×10^{1}	9.34×10^{0}	1.898×10^{1}	3
		S = 10	1.0382×10^2	5.42×10^{1}	4.539×10^{0}	9.390×10^{0}	3
8x8		Scatter	1.154×10^{2}	6.16×10^{1}	1.1082×10^{1}	2.2906×10^{1}	3
		S = 10	1.145×10^2	6.64×10^{1}	4.4600×10^{0}	9.651×10^{0}	3
12x6		Scatter	6.470×10^{1}	3.527×10^{1}	9.410×10^{0}	1.9141×10^{1}	3
		S = 10	6.265×10^{1}	3.417×10^{1}	4.555×10^{0}	9.495×10^{0}	3
16x8		Scatter	7.046×10^{1}	3.879×10^{1}	8.9535×10^{0}	1.8243×10^{1}	3
		S=10	6.70×10^{1}	3.854×10^{1}	5.239×10^{0}	1.0816×10^{1}	3

Performance as a function of grid shape and size, and S-parameter. "Best" performance is for the 12x6 grid with S=10.

the expense of slightly more factorization computational overhead and significantly more memory overhead (nearly a factor of two). Memory overhead could be reduced at the expense of further communication if explicit pivoting were used concommitantly.

Fourth, we can develop adaptive broadcast algorithms that track the known load imbalance in the B-mode factorization, and shift greater communication emphasis to nodes with less computational work remaining. For example, the pivot column is naturally a "hot spot" because the multiplier column (L column) must be computed before broadcast to the awaiting process columns. Allowing the non-pivot columns to handle the majority of the communication could be beneficial, even though this implies additional overall communication. Similarly, we might likewise apply this to

the pivot row broadcast, and especially for the pivot process, because it must participate in two broadcast operations.

We could utilize two process grids. When rows (columns) of U(L) are broadcast, extra broadcasts to a secondary process grid could reasonably be included. The secondary process grid could work on redistribution L/U to an efficient process grid shape and size for triangular solves while the factorization continues on the primary grid. This overlapping of communication and computation could also be used to reduce the cost of transposing the solution vector from column-distributed to row-distributed, which normally follows the triangular solves.

The sparse solver supports arbitrary user-defined pivoting strategies. We have considered but not fully

explored issues of fill-reduction vs. minimum time; in particular we have implemented a Markowitz-count fill-reduction strategy [4]. Study of the usefulness of partial column pivoting and other strategies is also needed. We will report on this in the future.

Reduced-communication pivoting and parametric distributions can be applied immediately to concurrent dense solvers with definite improvements in performance. While triangular solves remain lower-order work in the dense case, and may sensibly admit less tuning in S, the reduction of pivot communication is certain to improve performance. A new dense solver exploiting these ideas is under construction at present.

In closing, we suggest that the algorithms generating the sequences of sparse matrices must themselves be reconsidered in the concurrent setting. Changes that introduce multiple right-hand sides could help to amortize linear algebra cost over multiple time-like steps of the higher-level algorithm. Because of inevitable load imbalance, idle processor time is essentially free – algorithms that find ways to use this time by asking for more speculative (partial) solutions appear of merit toward higher performance.

Acknowledgements

The authors acknowledge Prof. Manfred Morari, who supervised the work presented in this student paper. We wish to acknowledge the dense concurrent linear algebra library provided by Eric Van de Velde, as well as a prototype sparse concurrent linear algebra library, both of which were useful springboards for this work.

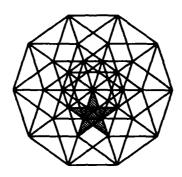
The first author acknowledges partial support under DOE grants DE-FG03-85ER25009 and DE-AC03-85ER40050. The second author (presently at the University of California, Santa Cruz) received support for his 1989 Caltech Summer Undergraduate Research Fellowship (SURF) under the same grants, and wishes to thank the Caltech SURF program for the opportunity to pursue the research discussed in part here.

The software implementation of this research was accomplished using machine resources made available by the Caltech Computer Science sub-Micron System Architectures Project and the Caltech Concurrent Supercomputer Facilities (CCSF).

References

[1] Alaghband, G, "Parallel pivoting combined with parallel reduction and fill-in control," *Parallel Computing* 11, 1989, pp. 201-221.

- [2] Athas W. C., and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, August 1988, pp. 9-24.
- [3] Brenan, K. E., S. L. Campbell, L. R. Petzold, Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, Elsevier, 1989.
- [4] Duff, I.S., A. M. Erisman and J. K. Reid, Direct Methods for Sparse Matrices, Oxford University Press, 1986.
- [5] Fox, G., et al., Solving Problems on Concurrent Processors, Volume 1, Prentice Hall, March 1988.
- [6] Golub, G. H., C. F. Van Loan, Matrix Computations, 2nd. Edition, John Hopkins University Press, 1989.
- [7] Skjellum, A., A. P. Leung, "Zipcode: A Portable Multicomputer Communication Library atop the Reactive Kernel," Proc. of DMCC5, Charleston, April 1990.
- [8] Skjellum, A., Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Differential-Algebraic Process Systems in Chemical Engineering, Ph.D. Dissertation, California Institute of Technology, Chemical Engineering, 1990.
- [9] Van de Velde, E. F., A Concurrent Direct Solver for Sparse Unstructured Systems, Caltech C³P Report #604, March 1988.
- [10] Van de Velde, E. F., The Formal Correctness of an LU-Decomposition Algorithm, Caltech C³P Report #625, June, 1988.
- [11] Van de Velde, E. F., Experiments with Multicomputer LU-Decomposition, Caltech / Rice Report CRPC-89-1. To appear in Concurrency: Practice and Experience.
- [12] Van de Velde, E. F., Adaptive Data Distribution for Concurrent Continuation, Caltech / Rice Center for Research in Parallel Computation Report CRPC-89-4. Submitted to Num. Math.



The Fifth Distributed Memory Computing Conference

13: Tridiagonal Systems

A Method to Parallelize Tridiagonal Solvers

Silvia M. Müller*

Computer Science Department (Bau 36)
University of Saarland
D-6600 Saarbrücken 11, FRG

Abstract:

We present a method to parallelize any tridiagonal solver, in a very efficient way. The communication overhead stays small. The parallel algorithms have nearly the same good qualities as their sequential counterparts, with respect to vectorization, speed and numerical aspects. The method is simple and independent of the sequential solver used. One yields some well known as well as many new parallel algorithms, when applied to standard sequential solvers.

Introduction:

In Numerical Mathematics there is a great interest in solvers for tridiagonal systems. By solving differential equations, tridiagonal systems with more than 10000 unknowns arise. There are many sequential solvers for such systems, like Gaussian Elimination, LU Factorization or Cyclic Reduction, and each solver has its preferences.

But often the sequential solvers are not fast enough. Parallel solvers are indispensible. It is desirable to have a parallel counterpart for each sequential solver. But how to get all these parallel algorithms? A simple method to parallelize all the algorithms would be very helpful. In the following sections we describe such a method in general and give some results.

Description of the method:

Propose that the number of processors p divides the order n of the system. Partitioning the original system A·x•d

$$A = \begin{bmatrix} a_1 & b_1 & & & \\ c_2 & a_2 & b_2 & & & \\ & & \ddots & \ddots & & \\ & & & & c_n & a_n \end{bmatrix}$$

into p subsystems, each processor works on n/p equations. The k - th system has the following form:

$$\begin{bmatrix} c_{j} & a_{j} & b_{j} & & & \\ c_{j+1} & a_{j+1} b_{j+1} & & & \\ & & \ddots & & \\ & & c_{r-1} a_{r-1} b_{r-1} & & \\ & & c_{r} & a_{r} & b_{r} \end{bmatrix} \times \begin{bmatrix} x_{j-1} \\ x_{j} \\ \vdots \\ \vdots \\ x_{r} \\ x_{r+1} \end{bmatrix} = \begin{bmatrix} d_{j} \\ \vdots \\ \vdots \\ d_{r} \end{bmatrix}$$

$$j = (k-1) \times n/p+1$$
 $r = k \times n/p$
 $c_1 = b_n = x_0 = x_{n+1} = 0$

Research partially funded by DFG, SFB 124

$$\begin{bmatrix} a_{j} & b_{j} & & & & \\ c_{j+1} & a_{j+1} b_{j+1} & & & \\ & ... & & & \\ c_{r-1} & a_{r-1} b_{r-1} & & & \\ & & c_{r} & a_{r} \end{bmatrix} \times \begin{bmatrix} x_{j} \\ \vdots \\ x_{r} \end{bmatrix} = \begin{bmatrix} d_{j} \\ \vdots \\ \vdots \\ d_{r} \end{bmatrix} - x_{j-1} \begin{bmatrix} c_{j} \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} - x_{r+1} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ b_{r} \end{bmatrix}$$

$$\begin{bmatrix} a_{j} & b_{j} \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ b_{r} \end{bmatrix}$$

$$\begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ b_{r} \end{bmatrix}$$

$$\begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ b_{r} \end{bmatrix}$$

$$\begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ b_{r} \end{bmatrix}$$

$$\begin{bmatrix} a_{j} & b_{j} \\ \vdots \\ \vdots \\ 0 \\ b_{r} \end{bmatrix}$$

The subsystems are neither quadratic nor independent. Each system, except the first and the last one, has two variables more than equations. Introducing these variables as parameters, the disturbing matrix elements vanish and for processor k system (1) results. Solving system (1) can also be seen as:

Solve one tridiagonal linear system with three different right hand sides

$$A^{k}u^{k} = d^{k}$$
 $A^{k}v^{k} = f^{k}$ $A^{k}z^{k} = g^{k}$ (2)

and choose x^k as a linear combination of the three partial solutions

$$x^{k} = u^{k} - x_{i-1}y^{k} - x_{i+1}z^{k}$$
 (3)

For each fixed k system (2) is quadratic and does not depend on other systems. Though each processor can solve its system (2) without any transfer, using an optional sequential solver. It suffices to convert the system into diagonal form and to change equation (3) a little bit:

$$a^{k} x^{k} = u^{k} - x_{j-1} y^{k} - x_{r+1} z^{k}$$
 (3')

a^k is the principle diagonal of the transformed system (2).

Before computing the global solution x^k the parameters, which connect the subsystems, must be determined. Taking the first and the last equation of each system (3') (only the last one of the first system and the first one of the last system) the tridiagonal system (4) of order 2p-2 arises. The two equations of processor k are:

$$a_{j}^{k} x_{j}^{k} * u_{j}^{k} - x_{j-1} y_{j}^{k} - x_{r+1} z_{j}^{k}$$

$$a_{r}^{k} x_{r}^{k} * u_{r}^{k} - x_{j-1} y_{r}^{k} - x_{r+1} z_{r}^{k}$$

System (4) can be solved in a sequential or in a parallel way. In order to solve the system

$$\begin{bmatrix} z_{n/p} & a_{n/p} \\ a_{n/p+1} & y_{n/p+1} & z_{n/p+1} \\ & & y_{2n/p} & z_{2n/p} & a_{2n/p} \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ &$$

in parallel, the above algorithm is used recursively, but with halved number of processors. For example, Gaussian Elimination could be used as solver. Each active processor solves a system of 4 equations with 3 right hand sides. After (log p -1) recursions only one processor is active and solves the remaining system sequentially.

Solving system (4) sequentially, data must be collected. After that transfer one processor computes the solution and sends the results to all other processors. The transfer can be executed in at most 2 log p steps.

The whole computation consists of three steps: First, each processor solves system (1) independent of all others, using an arbitrary sequential solver. This happens without any transfer.

Second, all processors together determine the parameters. They only need O(log p) transfer and a little extra computation.

Third, each processor computes the solution as a linear combination of the three partial solutions. This is also possible without any transfer.

Now it is easy, to give a coarse lower bound for the efficiency of these parallel algorithms.

- OM = number of all operations, whose result only depends on the matrix (sequential case)
- OR = number of all other <u>operations</u>
 especially those depending on the
 <u>right</u> hand side (sequential case)
- TP = number of all operations to solve system (4) and to transfer data (parallel case)

Efficiency = sequential runtime p + parallel runtime

Eff \geq (OR + OM) / (OM + 3 OR + TP · p).

OM and OR are linear in n and TP is linear in p in the worst case. If the dimension n is much bigger than the number of processors and the startup time, then eff $\geq 1/3$ (Let OM $\geq 3n$ and $p \leq 2^{10}$ then should be $p^2 \leq n$ and $p \leq n$). In reality the efficiency will be greater, because we have not used the special structure of both new right hand sides. So far the analysis of our method is independent of the sequential solver used.

Applications:

Applying the above method to Gaussian Elimination yields the well known Wang-Decompositon [1, 2]. Applying to Cyclic Reduction yields a parallel algorithm presented by GMD in March 1989 [2].

Now we want to give two examples and compare them with regard to efficiency and parallel runtime. This will be done not only for solving a tridiagonal system with one single right hand side, but as well for a system with several right hand sides successively fixed.

As <u>computational model</u> we use p processors interconnected by a crossbar, to determine runtimes. Floating point operations have cost A and a transfer with n data has cost S + n/B. S stands for startup time (time to build up a connection between the processors) and B for bandwidth. All other operations cost nothing.

Application to Gaussian Elimination:

The Wang - Decomposition is obtained, when applied to Gaussian Elimination.

Gaussian Elimination consists of two phases. During the first phase the entries below the principle diagonal are eliminated and during the second the remaining system is solved by backward elimination. It is easy to see that this can be done with 8n-7 floating point operations. For each new right hand side, which is fixed successively, the same number of operations is necessary.

Together with our method it is better to decompose the second phase of the Gaussian Elimination into two separate parts. During the first part of the second phase the entries above the principle diagonal are eliminated and during the second part the remaining system is solved. This means that system (1) is converted into diagonal form and equation (3') is used to determine the parameters and the linear combination.

Solving system (1) each processor has to treat three right hand sides. Making use of the special structure of the right hand sides processor k needs only 12 (n/p-1) A operations, as the following sequence illustrates:

During the first phase processor k computes the values

$$a_{i}^{k} = a_{i}^{k} - c_{i}^{k} / a_{i-1}^{k} b_{i-1}^{k}$$

$$d_{i}^{k} = d_{i}^{k} - c_{i}^{k} / a_{i-1}^{k} d_{i-1}^{k}$$

$$f_{i}^{k} = f_{i}^{k} - c_{i}^{k} / a_{i-1}^{k} f_{i-1}^{k} = -c_{i}^{k} / a_{i-1}^{k} f_{i-1}^{k}$$

$$g_{i}^{k} = g_{i}^{k} - c_{i}^{k} / a_{i-1}^{k} g_{i-1}^{k} = g_{i}^{k}$$

for each $i = j+1 \dots r$ and during the second

phase:

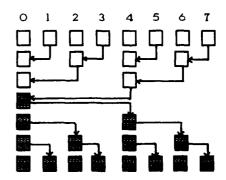
$$d_{i}^{k} = d_{i}^{k} - b_{i}^{k} / a_{i+1}^{k} d_{i+1}^{k}$$

$$f_{i}^{k} = f_{i}^{k} - b_{i}^{k} / a_{i+1}^{k} f_{i+1}^{k}$$

$$g_{i}^{k} = g_{i}^{k} - b_{i}^{k} / a_{i+1}^{k} g_{i+1}^{k} = -b_{i}^{k} / a_{i+1}^{k} g_{i+1}^{k}$$

for each $i-r-1 \dots j$.

Solving system (4) in parallel, each processor therefore needs (46 A+2S+12/B) $\log p - 21$ A operations. The data flow for this step is visualized in the following picture for p = 8.



- solving system (1) (origional or reduced system)
- computing linear combination (3')
- solving system (4) in the last recursion on one processor transfer

To determine solution x according to equation (3'), each processor needs 5(n/p-2) floating point operations. The first and the last value of x^k have already been computed during the previous steps. Altogether we get the following parallel runtime:

T_p = (17 n/p - 43) A + log p (46 A + 12/B + 25)
For each new right hand side fixed successively, the operations on the vectors f and g are the same and can be dropped. Thus only

(13 n/p - 39) A + log p (46 A + 12/B + 25) operations are necessary for a new right hand side.

Application to LU Factorization:

LU Factorization is the quickest direct solver for tridiagonal system. The algorithm is well suited for systems with more than one right hand side, even if they are fixed successively. First the matrix is factorized $A = L \cdot U$ and then two bidiagonal systems are solved.

$$A x = d \Rightarrow L (Ux) = d$$
$$= Ly = d & Ux = y$$

L is a lower and U an upper bidiagonal matrix. In the sequential case 8n-7 floating point operations are necessary to solve a system with one right hand side, and only 5n - 4 operations for each additional right hand side. Therefore it is interesting to have a look at the parallel version (this seems to be a new algorithm). To solve system (1) means now that each processor k factorizes Ak and solves the two bidiagonal systems for its three right hand sides. The second bidiagonal system is only converted into diagonal form and for the following steps equation (3') is used. For the factorization each processor needs 3 (n/p-1) floating point operations. L has only units on its principle diagonal. Together with the sparse structure of fk and gk, this fact leads to the amount of 9 (n/p-1) floating point operations to solve the factorized system (1). The second and the third step can be copied from the Wang Decomposition. Thus the following parallel runtime results:

 $T_p = (17 \text{ n/p} - 43) \text{ A} + \log p (46 \text{ A} + 12/\text{B} + 2S)$ For each new right hand side fixed successively, the factorization of the matrix A and the operations on the vectors f and g can be dropped. Thus for a new right hand side only

Comparison of these both parallel algorithms:

The algorithms are compared with respect to runtime and efficiency. We give theoretical results as well as measured values. For theory the following parameters are used:

$$A = 1$$
, $S = 4$, $B = 2$.

The parallel computer used is a PARSYTEC Megaframe with PAR_C. The foating point rate for double precision numbers is very small. Therefore the system has the parameters:

$$A = 1e$$
, $S = 4e$, $B = 2/e$, $e = 10^{-5}s$.

The number of processors and the granularity are significant for the quality of the algorithms.

Considering a system with one single right hand side:

Theoretical results:

parallel runtimes for Wang - Decomposition & LU Factorization

efficiency (%) for Wang - Decomposition & LU Factorization

Measured results:

G p	4	8	16	
2000	45,2	44,9	44.8	
4000	45.8	45.6	45.3	

efficiency (%) for Wang - Decomposition & LU Factorization

Considering a system with several right hand sides:

A
$$x^{t+1} = B x^t$$
, $t = 0 ... \tau$
A, B are tridiagonal systems

Such problems arise e.g. by solving PDE's by the finite difference method [3].

Theoretical results: (G = 4000, τ = 1000)

Measured results: (G = 4000, $\tau = 1000$)

n	16000	32000	64000		
Ga	2220	4587	9387		
LU	1709	3423	6844		
sequential runtimes (unit:lsec)					

þ	4	8	16		
par. Ga	783	809	831		
par. LU	646	650	653		
parallei runtimes (unit:lsec)					

p	4	8	16	
par. Ga	70,88	70,8	70.6	
par. LU	66,0	65,8	65,6	
efficiency (%) for $\tau = 1000$				

Conclusion:

The method can be applied to any solver and though we obtain the desired parallel version. It is also possible to generalize the method for systems with more than three diagonals. For m diagonals, m - 1 additional right hand sides are required. If k stays small the algorithms are efficient.

References:

- [1] Gannon, D. B. & Van Rosendale, J. (1984)

 On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms, In: IEEE TOC, vol. c 33,

 No. 12, pp 1180 1194.
- [2] Krechel, A. Plum, H. J. & Stüben, K. (1989)

 Solving Tridiagonal Linear Systems in

 Parallel on Local Memory MIMD Machines, Arbeitspapier der GMD 372, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin (FRG)
- [3] Todd, J. (1962) Survey of Numerical Analysis, Mc Graw Hill Book Company, INC, New York, pp. 419.

Solution of Periodic Tridiagonal Linear Systems on a Hypercube

Thiab R. Taha

Department of Computer Science University of Georgia Athens, Georgia 30602

Abstract

Periodic tridiagonal linear systems of equations typically arise from discretizing second order differential equations with periodic boundary conditions. Various vector algorithms are employed in order to solve such systems, namely: (i) A sweeping technique, (ii) Cyclic reduction, and (iii) LU decomposition methods. Implementations of the above methods are carried out on a vector extension board of an Intel iPSC/2 hypercube. Comparisons of the execution times of the utilized methods for solving different sizes of periodic tridiagonal systems are obtained.

1. Introduction:

Periodic banded systems of equations typically arise from discretizing second or higher order differential equations subjected to periodic boundary conditions [1,2]. In recent years, there has been a lot of research in developing algorithms for solving banded, and in particular tridiagonal systems on SIMD machines [9]. In this paper, three vector methods are employed in order to solve periodic tridiagonal linear systems which arise from discretizing second order differential equations with periodic boundary conditions, namely: (i) a sweeping technique, (ii) cyclic odd-even reduction, and (iii) LU decomposition methods.

Implementations of the above methods are carried out on a vector extension board of an Intel iPSC/2 hypercube. Comparisons of the execution time of the utilized methods for solving different sizes of periodic tridiagonal systems are obtained. The structure of this paper is as follows. We describe a test problem, the finite difference approximation to it, and the periodic system of equations which is a result of this approximation. We give a description of the methods and their vector implementations. We present the numerical results of the implementation on the iPSC/2 hypercube.

2. The test problem:

As an example a test problem is considered, namely;

$$-y'' + y = 2 Sinx$$
, $0 \le x \le 2\pi$ (1)
 $y(0) = y(2\pi)$,
 $y'(0) = y'(2\pi)$.

Eq. (1) can be discretized by a finite-difference method. We seek an approximation u_i to the $y(x_i)$, where $x_i = (i-1)h$, and h is the increment in x. Eq. (1) may be approximated by

$$-\frac{(u_{i+1}-2u_i+u_{i-1})}{h^2}+u_i=2\,Sinx_i\qquad (2)$$

which can be written as

$$-u_{i-1} + (2 + h^2)u_i - u_{i+1} = B_i$$
 (3)

where $B_i = 2h^2 Sinx_i$, $1 \le i \le N$. Using the periodic boundary conditions, Eq. (3) can be written in the matrix form as:

$$\begin{bmatrix} \alpha - 1 & -1 \\ -1 & \alpha - 1 & & \\ & \ddots & \ddots & \\ & & \ddots & \ddots & \\ & & & -1 & \alpha & -1 \\ -1 & & & -1 & \alpha \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{n-1} \\ B_n \end{bmatrix}, (4)$$

where $\alpha = 2 + h^2$.

3. Numerical methods for solving the test problem.

i. A Sweeping Technique:

Eq. (3) can be solved by a version of the Crank-Nicolson back and fourth sweep method

for the heat equation [1]. We seek an equation of the form

$$u_{i+1} = a u_i + b_i \tag{5}$$

suitable for computing u explicitly by sweeping to the right. For stability we require $|a| \le 1$. Repeated substitution of Eq. (5) into Eq. (3) to eliminate u_{i+1} and u_i in favor of u_{i-1} gives

$$b_i + b_{i-1} (a - (2 + h^2))$$

+ $u_{i-1} [a^2 - (2 + h^2)a + 1] = -B_i$ (6)

Requiring the u_{i-1} term to drop out determines a (uniquely since $|a| \le 1$) as a solution of

$$a^2 - (2 + h^2)a + 1 = 0 (7)$$

The two roots of Eq. (7) are
$$1 + \frac{h^2}{2} \mp \frac{h}{2} \sqrt{4 + h^2}$$
.

Requiring $a = 1 + \frac{h^2}{2} - \frac{h}{2} \sqrt{4 + h^2}$. From Eq.'s (6) and (7) we get

$$b_{i-1} = ab_i + a B_i \tag{8}$$

It follows that the b's can be computed explicitly by sweeping to the left. To obtain the u's, first solve for the b's from Eq. (8) then use Eq. (5) to calculate the u's. In order to calculate the b's we use an iteration procedure. We assume

$$b_{N+1} = constant value$$

to start with and then we apply the Gauss-Seidel technique [3] (in which the improved values are used as soon as they are computed) to calculate the rest of the b's. The calculated value of the $b_1(=b_{N+1})$ is used to start the new iteration, and the iteration procedure is repeated until the condition

$$|b_1 - b_{N+1}| < tolerance$$

is satisfied. Then we use the above procedure by sweeping to the right by means of Eq. (5) to obtain the u's. This method is well suited for serial computers but not for pipeline or vector computers due to the recursive nature of Eq.'s (5)

and (8). In this paper, the cyclic reduction method, to be discussed later, is used to solve the bidiagonal linear systems with a nonzero element on the lower left hand corner and a non zero element on the upper right hand corner generated from Eq.'s (8) and (5) respectively. It is to be noted that since the boundary conditions are periodic then $b_1 = b_{N+1}$ and $u_1 = u_{N+1}$. The systems of equations to be solved given in the matrix forms are:

$$\begin{bmatrix} 1 & -a & & & & \\ & 1 & -a & & & \\ & & 1 & & & \\ & & & \ddots & & \\ -a & & & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{bmatrix} = \begin{bmatrix} R_2 \\ R_3 \\ \vdots \\ R_n \\ R_{N+1} \end{bmatrix}, R_n = a B_n,$$
 (9)

and

$$\begin{bmatrix} 1 & & & -a \\ -a & 1 & & & \\ & -a & 1 & & \\ & & \cdot & \cdot & \\ & & & -a & 1 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_n \\ u_{N+1} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_N \end{bmatrix}$$
(10)

ii. Cyclic reduction method [5-8]

In this paper, the cyclic odd-even reduction method [6] is adapted in order to solve a class of periodic tridiagonal linear systems such as the one given in Eq. (4). This method is related to the cyclic reduction algorithm developed for the numerical solution of Possion's equation on a rectangle by Hockeny [7]. The main idea of this algorithm is the elimination of the odd-even unknowns and their eventual recovery through back substitution, i.e., this algorithm generates a sequence of problems $A^{(i)}X^{(i)} = b^{(i)}$ such that $X_j^{(i+1)} = X_{2j}^{(i)}$, and then recovers the odd-indexed unknowns of $X^{(i)}$ given $X^{(i+1)}$ through back substitution. For convenience the system in Eq.

(4) can be written in a general form as AX = b, where

$$X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{n-1} \\ X_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Suppose $A = (e_j, d_j, f_i)_{N \times N}$, $e_1 = f_N = 0$, and $N = 2^{m+1}$. Setting $N_i = 2^{m-i+1}$, we let $A^{(i)} = (e_j^{(i)}, d_j^{(i)}, f_j^{(i)})_{N_l \times N_l}$, $X^{(i)} = (X_j^{(i)})_{N_l}$, with $A^{(0)} = A, X^{(0)} = X, b^{(0)} = b$, and $\alpha^{(0)} = \alpha$. Then, for i = 0, ..., m-1 and $j = 1, ..., N_{i+1}$, define

$$\begin{split} d_{2j-1}^{(i)} &= \frac{1}{d_{2j-1}^{(i)}} , \\ e_{2j-1}^{(i)} &= d_{2j-1}^{(i)} e_{2j-1}^{(i)} , \\ f_{2j-1}^{(i)} &= d_{2j-1}^{(i)} f_{2j-1}^{(i)} , \\ b_{2j-1}^{(i)} &= d_{2j-1}^{(i)} b_{2j-1}^{(i)} , \\ \alpha^{(i)} &= d_{1}^{(i)} \alpha^{(i)} , \end{split}$$

and for $j = 1, ..., N_{i+1} - 1$ define

$$\begin{split} e_{j}^{(i+1)} &= -e_{2j}^{(i)} e_{2j-1}^{(i)} \;, \\ d_{j}^{(i+1)} &= d_{2j}^{(i)} - e_{2j}^{(i)} f_{2j-1}^{(i)} - f_{2j}^{(i)} e_{2j+1}^{(i)} \;, \\ f_{j}^{(i+1)} &= -f_{2j}^{(i)} f_{2j+1}^{(i)} \;, \\ X_{j}^{(i+1)} &= X_{2j}^{(i)} \;, \\ b_{j}^{(i+1)} &= b_{2j}^{(i)} - e_{2j}^{(i)} b_{2j-1}^{(i)} - f_{2j}^{(i)} b_{2j+1}^{(i)} \end{split}$$

and

$$\alpha^{(i+1)} = -\alpha^{(i)} e_2^{(i)}$$
,

$$e_{N_{in}}^{(i+1)} = -e_{N_i}^{(i)} e_{N_{i-1}}^{(i)},$$

$$d_{N_{in}}^{(i+1)} = d_{N_i}^{(i)} - \beta \alpha^{(i)} - e_{N_i}^{(i)} f_{N_{i-1}}^{(i)},$$

$$b_{N_{in}}^{(i+1)} = b_{N_i}^{(i)} - \beta b_{1}^{(i)} - e_{N_i}^{(i)} b_{N_{i-1}}^{(i)},$$

$$\beta = -\beta f_{1}^{(i)}$$

It is to be noted that the reduced system has the same form as A, and only the even-indexed unknowns appear. The reduction is continued until we have $A^{(m)}X^{(m)} = b^{(m)}$, a block 2×2 system which is solved by Gaussian elimination. For the backward substitution step, $X^{(i)}$ is found by

$$X_{2j}^{(i)} = X_{j}^{(i+1)}, \quad j = 1, ..., N_{i+1}$$

$$X_{1}^{(i)} = b_{1}^{(i)} - \alpha^{(i)} X_{N_{i}}^{(i)} - f_{1}^{(i)} X_{2}^{(i)}$$

$$X_{2j-1}^{(i)} = b_{2j-1}^{(i)} - e_{2j-1}^{(i)} X_{2j-2}^{(i)}$$

$$- f_{2i-1}^{(i)} X_{2j}^{(i)}, j = 2, ..., N_{i+1}.$$

iii. LU decomposition [4,5]

In this algorithm we assume that the LU decomposition of A exists; that is, A = LU where

$$L = \begin{bmatrix} 1 & & & & & \\ l_2 & 1 & & & & \\ & l_3 & 1 & & & \\ & & \ddots & & & \\ & & \vdots & \vdots & \vdots & \vdots \\ g_1 & g_2 & g_{n-2} & g_{n-1} & 1 \end{bmatrix}, \qquad (12)$$

$$U = \begin{bmatrix} u_1 & f_1 & & h_1 \\ u_2 & f_2 & & h_2 \\ & \ddots & & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & u_{n-1} & f_{n-2} & h_{n-2} \\ & & & u_{n} & h_{n-1} & \\ & & & & u_n \end{bmatrix}$$

with

$$u_1 = d_1, h_1 = \alpha,$$
 (12.1)

$$u_i = d_i - e_i \frac{f_{i-1}}{u_{i-1}}, i = 2, ..., n - 1$$
 (12.2)

$$l_i = \frac{e_i}{u_{i-1}}, i = 2, ..., n$$
 (12.3)

$$h_i = -l_i \ h_{i-1}, \ i = 2, ..., n-1$$
 (12.4)

$$h_{n-1} = h_{n-1} + f_{n-1}, (12.5)$$

$$g_1 = \frac{\beta}{\mu_1} \,, \tag{12.6}$$

$$g_i = -g_{i-1} \frac{f_{i-1}}{u_i}, i = 2, ..., n - 1$$
 (12.7)

$$g_{n-1} = g_{n-1} + l_n , \qquad (12.8)$$

$$u_n = d_n - \sum_{i=1}^{n-1} g_i h_i , \qquad (12.9)$$

The system LUx = b is then solved by the forward substitution Ly = b, which has the form

$$y_1 = b_1$$
,
 $y_i = b_i - l_i y_{i-1}$, $i = 2, ..., n - 1$ (13)
 $y_n = b_n - \sum_{i=1}^{n-1} g_i y_i$

followed by the back substitution Ux = y, which gives

$$x_{n} = \frac{y_{n}}{u_{n}},$$

$$y_{i} = y_{i} - h_{i} x_{n}, i = 1, ..., n - 1,$$

$$x_{n-1} = \frac{y_{n-1}}{u_{n-1}},$$

$$x_{i} = (y_{i} - f_{i} x_{i+1})/u_{i}, i = n-2, ..., 1$$
 (14)

The above algorithm is well suited for serial machine, but not for vector machines due to the recursive nature of Eq's. (12.2), (12.7), (13), and (14).

In this paper we consider an implementation of the LU method which is suitable for vector and pipeline computers. This implementation is given

by:

$$u_{1} = d_{1}, h_{1} = \alpha,$$

$$t_{i} = e_{i} f_{i-1}, \quad i = 2, ..., n - 1$$

$$u_{i} = d_{i} - \frac{t_{i}}{u_{i-1}}, \quad i = 2, ..., n - 1$$

$$u_{i} = \frac{1}{u_{i}}, \quad i = 1, ..., n$$

$$l_{i} = e_{i} u_{i-1}, \quad i = 2, ..., n$$

 $h_i = -l_i h_{i-1}, i = 2, \ldots, n-1$

$$h_{n-1} = h_{n-1} + f_{n-1}$$

$$g_1 = \beta u_1$$

 $z_i = f_{i-1} u_i, \quad i = 2, ..., n-1$
 $g_i = -g_{i-1} z_i, \quad i = 2, ..., n-1$

$$g_{n-1} = g_{n-1} + l_n$$

$$u_n = \frac{1}{d_n - \sum_{i=1}^n g_i h_i}$$

 $y_1 = b_1,$

Eq. (13) is solved by using the Veclib routine dlbidi on the intel iPSC/2 and can also be solved by the cyclic reduction method for bidiagonal systems.

$$y_{n} = b_{n} - \sum_{i=1}^{n-1} g_{i} y_{i},$$

$$x_{n} = y_{n} u_{n}$$

$$y_{i} = y_{i} - h_{i} x_{n}, \quad i = 1, ..., n-1$$

$$x_{i} = y_{i} u_{i}, \quad i = 1, ..., n-1$$

$$f_{i} = f_{i} u_{i}, \quad i = 1, ..., n-2$$

$$x_{i} = x_{i} - f_{i} x_{i+1}, \quad i = n-2, ..., 1 \quad (15)$$

Eq. (15) is solved by calling the Veclib routine dubidi and can also be solved by the cyclic

reduction algorithm for bidiagonal systems.

4. Numerical experiments and results

All of the three algorithms are implemented on the vector extension board of an Intel iPSC/2 hypercube. It is found that the sweeping technique, using the cyclic reduction method to solve the bidiagonal systems, is the most efficient one, followed by the cyclic reduction and finally by the LU decomposition. It is found that the sweeping technique is about 1.13 times faster than the cyclic reduction and about 2.2 times faster than the LU decomposition. Also, it is found that the sweeping vector method runs about 10 times faster than its serial version, the cyclic reduction is about 8 times faster than its serial version, and the LU decomposition is about 3 times faster than its serial version,

Acknowledgements:

This research has been supported in part by the U. S. Army Research Office and the National Science Foundation Grant No. CCR-8717033.

References

- [1] Taha, T.R. & Ablowitz, M.J. (1984)
 Analytical and Numerical Aspects of Certain
 Nonlinear Evolution Equations. II.
 Numerical, Nonlinear Schrödinger Equation,
 J. Comput. Phys. 55, pp. 203-230.
- [2] Taha, T.R. & Ablowitz, M.J. (1984) Analytical and Numerical Aspects of Certain Nonlinear Evolution Equations, III. Numerical, Korteweg-de Vries Equation, J. Comput. Phys. 55, pp. 231-253.
- [3] Smith, G.D. (1965) Numerical Solution of Partial Differential Equations. Oxford Univ. Press, New York.
- [4] Lambiotle Jr., J.J. & Voigt, R.G. (1975) The Solution of Tridiagonal Linear Systems on the CDC Star-100 Computer, ACM Trans. Math. Software 1 (4), pp. 308-329, December.
- [5] Zhong, L. (1989) A Comparison of Parallel Algorithms for the Solution of Tridiagonal Linear Systems of Equations, MAMS Technical Report, University of Georgia.
- [6] Heller, D. (1976) Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems, SIAM, J. Numer. Anal. 13

- (4) pp. 484-496.
- [7] Hockney, R.W. (1965) A Fast Direct Solution of Poisson's Equation Using Fourier Analysis, J. Assoc. Comput. Mach. 12, pp. 95-113.
- [8] Kershaw, D. (1982) Solution of Single Tridiagonal Linear Systems and Vectorization of the ICCG Algorithm on the Cray-1, In: Garry Rodrigue (ed.): Parallel Computations, Academic Press, pp. 85-99.
- [9] Gallivan, K., Plemmons, R., & Sameh, A. (1990) Parallel Algorithms for Dense Linear Algebra Computations, SIAM, Rev. 32, 1, pp. 54-135.

THE ERROR ANALYSIS OF A TRIDIAGONAL SOLVER

Hong Zhang

Department of Mathematical Sciences Clemson University Clemson, SC 29634-1907

ABSTRACT

The Parallel Diagonal Dominant (PDD) Algorithm has been proposed for solving certain types of tridiagonal linear systems. The algorithm employ a matrix approximation. Both theoretical and experimental results have shown that the PDD algorithm is a highly efficient parallel algorithm for a variety of architectures. In this paper, the effect of this approximation is studied and a rigorous error analysis is given. The numerical results are presented.

Introduction

The Parallel Diagonal Dominant (PDD) Algorithm has been proposed in [1] for solving tridiagonal linear systems. The algorithm is based on the divide & conquer model of parallel computation. First, a linear system of order n is divided into p subsystems which can be solved concurrently by p processors. Then the subsolutions are modified by the solution of a conquer system of order 2(p-1) to obtain the solution of the original system. Under certain assumptions, the coefficient matrix of this 2(p-1)-dimensional conquer system converges to a diagonal block matrix at least exponentially as $n/p \to \infty$. This indicates that we can take this diagonal block matrix instead if $n \gg p$. It reduces the communication cost and the algorithm almost reaches ideal p speedup if p processors are used. Both theoretical and experimental results have shown that the PDD algorithm is a highly efficient parallel algorithm for a variety of architectures. In this paper we study the accuracy of the results of the PDD algorithm, discuss how the solution of the system is affected by the matrix approximation and give a rigorous error analysis.

Section 1 briefly describes the PDD algorithm. Section 2 derives the error bound. Computational results are presented in Section 3.

1. The parallel diagonal dominant(PDD) algorithm

The PDD algorithm is used to solve the linear system of the form

$$\mathbf{A} \mathbf{x} = \mathbf{d} , \qquad (1.1)$$

where

$$A = (a_i, b_i, c_i)$$
 $(i = 0,..., n-1)$ (1.2)

is an $n \times n$ tridiagonal matrix satisfying certain conditions. For convenience we assume that n = pm. A tridiagonal matrix A is called *evenly diagonal dominant* if

$$|a_i| \le |b_i/2|, |c_i| \le |b_i/2| \text{ and } a_{i+1} \cdot c_i > 0.$$
 (1.3)

Suppose matrix A in (1.1) is evenly diagonal dominant. Scaling both sides of (1.1), without loss of generality, we assume that matrix $A = (a_i, b_i, c_i)$ satisfies

$$|a_i| \le 1, |c_i| \le 1, b_i \ge 2,$$

and

$$a_{i+1} \cdot c_i > 0. \tag{1.4}$$

Matrix A can be written as

$$\mathbf{A} = \tilde{\mathbf{A}} + \Delta \mathbf{A} \tag{1.5}$$

with

The submatrices $A_j = (a_i^{(j)}, b_i^{(j)}, c_i^{(j)})$ are $m \times m$ matrices. Let e_i be a column vector with its ith $(0 \le i \le n-1)$ entry being 1 and all the other entries being zero. The ΔA can be expressed as $\Delta A = VE^T$, with

$$\mathbf{V} = \left[a_m \, \mathbf{e}_m \, , \, c_{m-1} \mathbf{e}_{m-1} \, , \, a_{2m} \, \mathbf{e}_{2m} \, , \, \cdots \, , \, c_{(p-1)m-1} \mathbf{e}_{(p-1)m-1} \right]$$

$$\mathbf{E} = \left[\mathbf{e}_{m-1}, \, \mathbf{e}_{m}, \, \dots, \, \mathbf{e}_{(p-1)m-1}, \, \mathbf{e}_{(p-1)m} \, \right],$$

both V and E are $n \times 2(p-1)$ matrices. Based on the matrix modification formula[2], the solution of (1.1) is

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{d} = (\tilde{\mathbf{A}} + \mathbf{V} \mathbf{E}^{T})^{-1}\mathbf{d}$$
$$= \tilde{\mathbf{A}}^{-1}\mathbf{d} - \tilde{\mathbf{A}}^{-1}\mathbf{V}(\mathbf{I} + \mathbf{E}^{T}\tilde{\mathbf{A}}^{-1}\mathbf{V})^{-1}\mathbf{E}^{T}\tilde{\mathbf{A}}^{-1}\mathbf{d}. \tag{1.6}$$

Introducing a permutation matrix P, the band width of the matrix $I+E^T \tilde{A}^{-1}V$ can be reduced from 5 to 3. The solution of (1.1) then becomes

$$\mathbf{x} = \tilde{\mathbf{A}}^{-1}\mathbf{d} - \tilde{\mathbf{A}}^{-1}\mathbf{VPZ}^{-1}\mathbf{E}^{T}\tilde{\mathbf{A}}^{-1}\mathbf{d}, \qquad (1.7)$$

where $\mathbf{Z} = \mathbf{P} + \mathbf{E}^T \tilde{\mathbf{A}}^{-1} \mathbf{V} \mathbf{P}$ is a $2(p-1) \times 2(p-1)$ tridiagonal matrix. The equation (1.1) is solved in the following steps:

- 1. Solve $\tilde{\mathbf{A}} \tilde{\mathbf{x}} = \mathbf{d}$. $\tilde{\mathbf{A}} \mathbf{Y} = \mathbf{VP}$.
- 2. Form $h = E^T \bar{x}$, $Z = P + E^T Y$.
- 3. Solve $\mathbf{Z} \mathbf{y} = \mathbf{h}$.
- 4. Compute $\Delta x = Yy$, $x = \tilde{x} \Delta x$.

Since matrix $\tilde{\mathbf{A}}$ is a diagonal block matrix, step 1 can be done concurrently. It is equivalent to solve three subsystems of order m with same coefficient matrix. There is no computation at step 2. Due to the special structure of matrix \mathbf{Y} , step 4 can be computed by p processors simultaneously. Different ways of solving the system at step 3 result in different algorithms. Let us denote the matrix

$$Z = (a_i^z, b_i^z, c_i^z)$$
 $(i = 0,..., 2(p-1)-1)$ (1.8)

and the determinants of the submatrices A_i as $det(A_i)$

(j=0,...,p-1). It is proved in [1] that

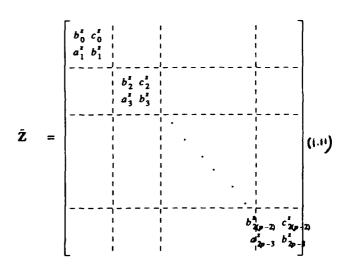
$$|a_{2j}^{x}| = \prod_{i=0}^{m-1} |a_{i}^{(i)}| / \det(A_{j}),$$

$$|c_{2j-1}^{z}| = \prod_{i=0}^{m-1} |c_{i}^{(j)}| \det(\mathbf{A}_{j}), \qquad (1.9)$$

for j = 1,..., p-2 and

$$max(|a_{2j}^z|,|c_{2j-1}^z|) \le \frac{1}{(m+1)(1+\varepsilon/4)^m}$$
 (1.10)

with $\varepsilon = \min_{0 \le j \le n-1} (b_i - 2)$. When $\varepsilon > 0$, the inequality (1.10) shows that the off-diagonal elements a_{2j}^{ϵ} , c_{2j-1}^{ϵ} (j = 1,..., p-2) of **Z** converge to zero at least exponentially as $m = n/p \to \infty$. From this result, the PDD algorithm uses matrix



instead of \mathbb{Z} when $n \gg p$ at step 3. This matrix approximation removes the bottleneck of the computation and makes the algorithm highly parallel. It also reduces the communication cost into two neighboring communications only. Both theoretical and experimental results have shown that, using p processors, the PDD algorithm almost reaches p speed-up when the matrix condition is permitted. For the detailed description of the PDD algorithm, the reader may refer to [1].

2. The accuracy of the PDD algorithm

The PDD algorithm uses the approximate matrix $\tilde{\mathbf{Z}}$ instead of \mathbf{Z} at step 3. It then raises the question about the accuracy of the result. It has been claimed in [1] that if $\tilde{\mathbf{Z}}$

equals Z within machine accuracy, the error between the approximate and the exact solutions would have the same order of magnitude. In this section we study the effect of the matrix approximation in detail and give a rigorous error analysis.

For simplicity we assume that the evenly diagonal dominant matrix $A = (a_i, b_i, c_i)$ in (1.1) satisfies (1.4) with

$$\varepsilon = \min_{0 \le i \le n-1} (b_i - 2) > 0. \tag{2.1}$$

Define $\Delta Z = Z - \tilde{Z}$, i.e.,

$$\Delta Z = \begin{bmatrix} c_1^z & c_2^z & c_3^z & c_3^$$

It is easy to verify that

$$|\Delta Z| = \max_{1 \le j \le p-2} (|a_{2j}^z|, |c_{2j-1}^z|)$$
 (2.3)

$$\leq \frac{1}{(m+1)(1+\varepsilon/4)^m}$$
, from (1.10).

Here and throughout, $\| \| \|$ denotes the spectral matrix norm[3, pg.81]. Let $y + \Delta y$ be the exact solution of the system

$$\tilde{\mathbf{Z}}(\mathbf{y} + \Delta \mathbf{y}) = \mathbf{h} \,, \tag{2.4}$$

then $\bar{Z}\Delta y = \Delta Z y$ because Z y = h (see step 3 of section 2). It has been proved[1] that \bar{Z} is invertible, thus

$$\|\Delta y\| \le \|\tilde{Z}^{-1}\| \|\Delta Z\| \|y\|.$$
 (2.5)

The relative error for the solution of $\mathbf{Z}\mathbf{y} = \mathbf{h}$ is bounded by $\|\mathbf{\tilde{Z}}^{-1}\|\|\Delta\mathbf{Z}\|$. An upper bound of $\|\mathbf{\tilde{Z}}^{-1}\|$ is given by

Proposition 2.1. Let $\tilde{\mathbf{Z}}$ be defined in (1.11). Then

$$\|\tilde{\mathbf{Z}}^{-1}\| \le \frac{3(1+\varepsilon)}{\varepsilon}$$
, (2.6)

with ε defined by (2.1).

Proof. $\tilde{\mathbf{Z}}$ is a diagonal block matrix. The jth block of $\tilde{\mathbf{Z}}$ is

$$\begin{bmatrix} b_{2j}^{z} & c_{2j}^{z} \\ a_{2j+1}^{z} & b_{2j+1}^{z} \end{bmatrix} = \begin{bmatrix} b_{2j}^{z} & 1 \\ 1 & b_{2j+1}^{z} \end{bmatrix} \quad (j = 0, ..., p-2).$$

The eigenvalues of the jth block are

$$\lambda_{\pm}^{(j)} = \frac{b_{2j}^z + b_{2j+1}^z \pm \sqrt{(b_{2j}^z + b_{2j+1}^z)^2 + 4(1 - b_{2j}^z b_{2j+1}^z)}}{2}.$$

Let $\sigma(\tilde{Z})$ be the spectrum of \tilde{Z} , then

$$\|\tilde{\mathbf{Z}}^{-1}\| = \frac{1}{\min |\lambda|} = \frac{1}{\min (|\lambda_{+}^{(j)}|, |\lambda_{-}^{(j)}|)}$$

$$\lim_{\lambda \in \operatorname{G}(\tilde{\mathbf{Z}})} \sup_{0 \le j \le \rho - 2} |\lambda_{+}^{(j)}| = \frac{1}{\min (|\lambda_{+}^{(j)}|, |\lambda_{-}^{(j)}|)}$$

It is sufficient to find the lower bound of $|\lambda_{\pm}^{(j)}|$. Suppose submatrices $A_j = (a_i^{(j)}, b_i^{(j)}, c_i^{(j)})$ (j=0,...,p-1) in (1.5) have LDU factorization

$$\mathbf{A}_{i} = (l_{i}^{(j)}, 1, 0)(0, \delta_{i}^{(j)}, 0)(0, 1, u_{i+1}^{(j)})$$

then (see [1, Appendix])

$$b_{2j}^{z} = \frac{c_{m-1}^{(j)}}{\delta_{m-1}^{(j)}}, \qquad (2.7)$$

$$b_{2j+1}^{z} = \left(\frac{1}{\delta_{0}^{(j+1)}} + \frac{c_{0}^{(j+1)}a_{1}^{(j+1)}}{(\delta_{0}^{(j+1)})^{2}\delta_{1}^{(j+1)}} + \cdots\right)$$

$$+\frac{c_0^{(j+1)}\cdots c_{m-2}^{(j+1)}a_1^{(j+1)}\cdots a_{m-1}^{(j+1)}}{(\delta_0^{(j+1)}\cdots \delta_{m-2}^{(j+1)})^2\delta_{m-1}^{(j+1)}})a_0^{(j+1)}. (2.8)$$

Using mathematical induction, equation (1.4) and the fact that $\delta_i^{(j)} = b_i^{(j)} - a_i^{(j)} c_{i+1}^{(j)} / \delta_{i-1}^{(j)}$, $\delta_0^{(j)} = b_0^{(j)}$, we obtain

$$\delta_i^{(j)} \ge \varepsilon + \frac{i+2}{i+1} \,. \tag{2.9}$$

It then follows that, for any r < m,

$$\delta_0^{(j)} \cdots \delta_r^{(j)} \ge (\varepsilon + \frac{2}{1})(\varepsilon + \frac{3}{2})(\varepsilon + \frac{4}{3}) \cdots (\varepsilon + \frac{r+2}{r+1}) \quad (2.10)$$

$$\geq (r+2) + \varepsilon M(r), \qquad (2.11)$$

where r+2 and M(r) are the constant and the coefficient of ε for the polynomial on the right hand side of (2.10). M(r) can be written as a finite sum

$$M(r) = \frac{3 \cdot 4}{2 \cdot 3} \cdot \frac{r+2}{r+1} + \frac{2 \cdot 4}{1 \cdot 3} \cdot \frac{5}{4} \cdot \frac{r+2}{r+1} + \cdots$$

$$= (r+2) \sum_{i=1}^{r} \frac{i}{i+1} + r+1. \tag{2.12}$$

Using (1.4) and (2.7)-(2.12), it yields

$$|b_{2j}^{z}| \leq \frac{1}{\delta_{m-1}^{(j)}} \leq \frac{1}{\varepsilon + \frac{m+1}{m}}$$

$$= \frac{m}{m\varepsilon + m + 1}, \qquad (2.13)$$

$$|b_{2j+1}^{z}| \leq \frac{1}{\delta_{0}^{(j+1)}} + \frac{c_{0}^{(j+1)}a_{1}^{(j+1)}}{(\delta_{0}^{(j+1)})^{2}\delta_{1}^{(j+1)}} + \frac{c_{0}^{(j+1)}\cdots c_{m-2}^{(j+1)}a_{1}^{(j+1)}\cdots a_{m-1}^{(j+1)}}{(\delta_{0}^{(j+1)}\cdots \delta_{m-2}^{(j+1)})^{2}\delta_{m-1}^{(j+1)}}$$

$$\leq \frac{1}{2+\varepsilon M(0)} + \frac{2}{(2+\varepsilon M(0))^{2}\cdot 3} + \frac{3}{(3+\varepsilon M(1))^{2}\cdot 4} + \cdots + \frac{m}{(m+\varepsilon M(m-2))^{2}(m+1)}$$

$$= \frac{1}{2+\varepsilon M(0)} + \sum_{i=2}^{m} \frac{i}{(i+\varepsilon M(i-2))^{2}(i+1)}. \qquad (2.14)$$

Let M_1 , M_2 be the expressions on the right hand sides of (2.13) and (2.14), we obtain

$$M_1 = \frac{m}{m\varepsilon + m + 1} < \frac{1}{\varepsilon + 1} \tag{2.15}$$

$$M_2 = \frac{1}{2 + \varepsilon M(0)} + \sum_{i=2}^{m} \frac{i}{(i + \varepsilon M(i-2))^2 (i+1)}$$
 (2.16)

$$<\frac{1}{2}+\sum_{i=2}^{m}\frac{1}{i(i+1)}=\frac{m}{m+1}<1.$$

Inequalities (2.13)-(2.16) imply that

$$b_{2j}^{z} b_{2j+1}^{z} < 1$$
 for j=0,...,p-2. (2.17)

From (2.7), (2.8) and (1.4), b_{2j}^z and b_{2j+1}^z have the same sign. Without loss of generality, they are assumed to be positive. Using (2.13)-(2,14),

$$\min \left(|\lambda_{+}^{(j)}|, |\lambda_{-}^{(j)}| \right) =$$

$$\frac{-(b_{2j}^{z} + b_{2j+1}^{z}) + \sqrt{(b_{2j}^{z} + b_{2j+1}^{z})^{2} + 4(1 - b_{2j}^{z} b_{2j+1}^{z})}}{2}$$

$$= \frac{2(1 - b_{2j}^{z}, b_{2j+1}^{z})}{(b_{2j}^{z} + b_{2j+1}^{z}) + \sqrt{(b_{2j}^{z} + b_{2j+1}^{z})^{2} + 4(1 - b_{2j}^{z} b_{2j+1}^{z})}}$$

$$\geq \frac{2(1 - M_{1} \cdot M_{2})}{(M_{1} + M_{2}) + \sqrt{(M_{1} + M_{2})^{2} + 4}}$$

$$\geq \frac{2(1 - \frac{1}{1 + \varepsilon})}{2 + \sqrt{2^{2} + 4}} > \frac{\varepsilon}{3(1 + \varepsilon)}.$$
(2.18)

The lower bound of $|\lambda_{\pm}^{(j)}|$ then yields the inequality (2.6). \square

Remark. Note that the lower bound (2.18) is much better than $\varepsilon/3(1+\varepsilon)$ except that the expression in (2.18) is more complex. Using (2.18) a better upper bound of $\|\tilde{\mathbf{Z}}^{-1}\|$ is

$$\|\ddot{\mathbf{Z}}^{-1}\| \le \frac{(M_1 + M_2) + \sqrt{(M_1 + M_2)^2 + 4}}{2(1 - M_1 \cdot M_2)},$$
 (2.19)

where M_1 , M_2 are defined by (2.13)-(2.14) and M(r) is defined in (2.12).

Suppose $n \gg p$, as we mentioned before, $\|\Delta Z\|$ can be as small as machine accuracy. Without loss of generality, we can assume that $\|\tilde{Z}^{-1}\| \|\Delta Z\| < 1$. Since $Z = \tilde{Z} + \Delta Z = \tilde{Z}(I + \tilde{Z}^{-1}\Delta Z)$, matrix Z is invertible and

$$\|\mathbf{Z}^{-1}\| = \|(\mathbf{I} + \bar{\mathbf{Z}}^{-1}\Delta\mathbf{Z})^{-1}\bar{\mathbf{Z}}^{-1}\|$$
 (2.20)

$$\leq \frac{\|\tilde{\mathbf{Z}}^{-1}\|}{1 - \|\tilde{\mathbf{Z}}^{-1}\|\|\Delta \mathbf{Z}\|}.$$

The absolute error $||Y\Delta y||$ for the original system Ax = d

niques, we may able to find better error bound.

$$\|\mathbf{Y}\Delta\mathbf{y}\| = \|\mathbf{\tilde{A}}^{-1}\mathbf{V}\mathbf{P}\Delta\mathbf{y}\|$$

$$\leq \mid\mid\!\mid\!\tilde{A}^{-1}\mid\mid\mid\mid\mid\!\Delta y\mid\mid\!\leq\mid\mid\!\mid\!\tilde{A}^{-1}\mid\mid\mid\mid\mid\!\tilde{Z}^{-1}\mid\mid\mid\mid\mid\!\Delta Z\mid\mid\mid\mid\mid\!Z^{-1}\mid\mid\mid\mid\mid\!\tilde{A}^{-1}d\mid\mid$$

by (2.5) and $y = Z^{-1}E^T \tilde{A}^{-1}d$. Combining (2.20) we obtain

$$\|\mathbf{Y}\Delta\mathbf{y}\| \leq \frac{\|\tilde{\mathbf{A}}^{-1}\|\|\tilde{\mathbf{Z}}^{-1}\|^2 \|\Delta\mathbf{Z}\|\|\tilde{\mathbf{A}}^{-1}\mathbf{d}\|}{1 - \|\tilde{\mathbf{Z}}^{-1}\|\|\Delta\mathbf{Z}\|}.$$
 (2.21)

The relative error consequently satisfies

$$\frac{\|\mathbf{Y}\Delta\mathbf{y}\|}{\|\mathbf{x}\|} \le \frac{\|\tilde{\mathbf{A}}^{-1}\|\|\tilde{\mathbf{Z}}^{-1}\|^2 \|\Delta\mathbf{Z}\|}{1 - \|\tilde{\mathbf{Z}}^{-1}\|\|\Delta\mathbf{Z}\|} \cdot \frac{\|\tilde{\mathbf{A}}^{-1}\mathbf{d}\|}{\|\mathbf{A}^{-1}\mathbf{d}\|}. \quad (2.22)$$

Let us assume that $\|\bar{\mathbf{A}}^{-1}\mathbf{d}\|$ and $\|\mathbf{A}^{-1}\mathbf{d}\|$ have about the same magnitude (i.e., $\frac{\|\bar{\mathbf{A}}^{-1}\mathbf{d}\|}{\|\mathbf{A}^{-1}\mathbf{d}\|} = O(1)$) which usually is true in practice. Using the fact $\|\bar{\mathbf{A}}^{-1}\| \le 1/\epsilon$, an approximate upper bound for the relative error is

$$\frac{\|\tilde{\mathbf{Z}}^{-1}\|^{2}}{\varepsilon (1-\|\tilde{\mathbf{Z}}^{-1}\|\|\Delta\mathbf{Z}\|)} \cdot \|\Delta\mathbf{Z}\| \qquad (2.23)$$

where the bound of $\|\tilde{\mathbf{Z}}^{-1}\|$ and $\|\Delta \mathbf{Z}\|$ are given by proposition 2.1 and (2.3) respectively.

The bound (2.23) indicates that if ε is not too small and $n \gg p$, the PDD algorithm offers a satisfactory solution.

3. Numerical Results

The experiments were conducted for the system A x = d with $A = (1, 2+\varepsilon, 1)$ and $d = (1, 1,..., 1)^T$. The computations were done in double precision. The numerical results and our estimates are listed in Table 1 and Table 2. We use (2.3) as the estimated $||\Delta Z||$ and (2.19), (2.23) as the estimated relative error for the solution of the system.

Both the theoretical analysis process and the numerical results indicate that the real computational errors can be much smaller than our estimation. Also as we mentioned in [1] that the assumption on the matrices is sufficient but not necessary. The algorithm can be applied to any positive definite or diagonal dominant matrix whose resulting elements $|a_{2j}^2|$, $|c_{2j-1}^2|$ (j=1,...,p-2) become underflow and $||\tilde{A}^{-1}|| ||\tilde{Z}^{-1}||^2$ is not too large. For the special matrices, using similar tech-

Table 1

	1 4	Z	Relative Error	
E	Computed	Estimated	Compused	Estimated
10-5	1.9×10 ⁻⁰³	2,5×10 ⁻⁶³	8.0×10 ⁻⁴²	3.1×10 ²
10~	3.6×10 ⁻⁰⁴	2.5×10 ⁻⁰⁰	3,2×10 ⁻⁰³	3.6×10 ¹
10-3	2.0×10 ⁻⁰⁷	2.3×10 ⁻⁶⁰	3.1×10 ⁻⁰⁷	4.5×10 ^a
10-2	7.8×10 ⁻¹⁹	9.2×10 ⁻⁰⁴	0.0	3.4×10 ⁻¹
10-1	9.2×10 ⁻³⁶	1.3×10 ⁻⁰⁷	0.0	2.4×10-05
1.0	5.5×10 ⁻¹⁴⁴	4.3×10 ⁻⁴²	0.0	4.1×10 ⁻⁴¹

n=6400, p=16, n/p=400

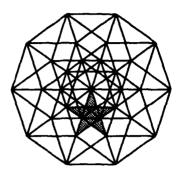
Table 2

	4	z	Relative Error	
	Computed	Estimated	Computed	Estimated
10-5	4.0×10 ⁻⁵	6.2×10 ⁻⁰⁴	1.9×10 ⁻⁶³	7.7×10¹
10→	2.2×10 ⁻⁰⁹	6.0×10 ⁻⁰⁴	2.0×10 ⁻⁰⁸	8.8×10°
10-3	6.5×10 ⁻²⁴	4.2×10 ⁻⁰⁴	0.0	8.4×10 ⁻⁶¹
10-2	6,3×10 ⁻⁷¹	1.1×10 ⁻⁰⁵	0.0	4.3×10 ⁻⁰³
10-1	6.9×10 ⁻²²⁰	4.3×ነሆ ²¹	0.0	8.1×10 ⁻¹⁹
1.0	0.0	5.5×10 ⁻¹⁵⁹	0.0	5.3×10 ⁻¹⁵⁸

n=6400, p=4, n/p=1600

References

- X. Sun, H. Z. Sun, and L. M. Ni, Parallel Algorithms for Solution of Tridiagonal Systems on Multicomputers, Proc. of the 1989 ACM International Conference on Supercomputing, (Crete, Greece, June 5-9, 1989).
- [2] I. S. Duff, A. M. Erisman, and J. K. Reid, Direct Methods for Sparse Matrices. Calrendon Press, Offord, (1986).
- [3] J. H. Wilkinson, Rounding Errors in Algebraic Processes. Prentice-Hall, Inc. (1963).



The Fifth Distributed Memory Computing Conference

14: Basic Algorithms

An Efficient FFT Algorithm on Multiprocessors with Distributed Memory

J. P. Zhu *

Department of Applied Mathematics State University of New York, Stony Brook, NY 11794, U.S. A.

January 1, 1990

Abstract

This paper discusses a parallel FFT algorithm and its implementation on iPSC/2 hypercube. Numerical experiment and performance model analysis show that parallel FFT algorithm can be implemented on hypercubes efficiently. The internode communication cost is minimized by eliminating fragmentary message passing, overlapping communication and computation and mapping all data groups that need mutual communication into the closest neighbors on a hypercube:

Introduction

FFT is one of the most widely used numerical method in science and engineering, especially in the area of signal and image processing, time series and spectral analysis, computational mechanics and the numerical solution of partial differential equations (such as solving Poisson's equation by fast direct method)[9]. For sequential FFT algorithm, there have been many developments and improvements during the past twenty years since FFT was first introduced by Cooley and Tukey (1965), see [2],[5], [6], [7], for details. Along with the development of vector computers, FFT algorithm has also been modified and implemented on computers with vector processing facilities, see [1], [10], [13]-[17]. Theoretical and numerical experiment analysis show that FFT is highly amenable to vector or SIMD architectures and significant improvements in performance have been achieved on this class of computers.

Recently, the problem of implementing FFT algorithm on multi-processor parallel computers with shared memory (MIMD machine) was investigated by Briggs, et al. (1987) and mathematical performance models were established to quantify the analysis of the algorithm performance. They demonstrated, through numerical experiment and theoretical analysis, that the implementation of FFT algorithm on shared memory parallel computers has been quite successful.

While discussing general problem solving strategies on parallel computers with distributed memory, Fox (1988) and Walker (1988) studied a parallel FFT algorithm on hypercube as an example. The algorithm is based on a group of general purpose message passing routines for internode communications. Once the communication routines are implemented on a target machine, the algorithm can be ported to the new machine without difficult.

The purpose of this paper is to investigate the implementation of FFT algorithm on iPSC/2 hypercube and present results of numerical experiments and performance model analyses. Both computation results and theoretical analyses show that the communication cost can be minimized by eliminating fragmentary message passing, overlapping communication and computation

^{&#}x27;The author thanks Cornell Theory Center for providing access of its iPSC/2 hypercube

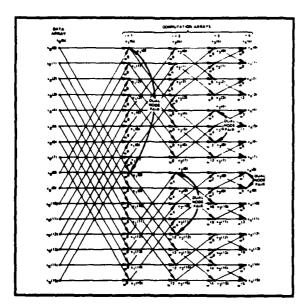


Figure 1: Flow chart of FFT

and mapping all data groups that need mutual communications into the closest neighbors on a hypercube.

The Algorithm:

There are many variants of FFT algorithm in the references. We started by introducing parallelism in the algorithm discussed in [4]. To illustrate the parallel algorithm, we use the following example.

For a given array of complex elements $X_0(k)$, $k=0,\cdots,15$, the final result of its FFT, denoted by $X_4(k)$, can be calculated in four steps as shown in Fig. 1. where $w^l = exp(2\pi i l/16)$, $i = \sqrt{-1}$. Each column in Fig.1 represents the array of intermediate result during the computation. Every point (except points in the first column) is an element in the array of intermediate result and is entered by two solid lines representing transmission paths from previous points. A solid line transmits or brings a value from a point in the previous step, multiplies the value by w^l , and inputs the result into the point in the next array. Factor w appears near the arrowhead, absence of this factor means $w^l = 1$. Results entering a node from the two transmission paths are combined additively. It is clear that in the array of each step, points can be classified into pairs, the two points in each pair has input transmission paths stemming from the same pair of points in the previous array. We call the two points in such a pair a dual point pair. Each dual point pair in $X_i(k)$, $i = 1, \dots, 4$ is calculated by using a corresponding dual pair in $X_{i-1}(k)$ and multiplication of a complex exponentiation. For example:

$$X_1(0) = X_0(0) + w^0 X_0(8)$$

$$X_1(8) = X_0(0) + w^8 X_0(8)$$

$$= X_0(0) - w^0 X_0(8)$$
(1)

Other pairs can be obtained in similar manner. The index difference between points in dual pair (originally 8) is halved in every step until it reaches 1 in last step, which means the dual pair now contains adjacent elements. In each intermediate step, the computations for different dual pairs are independent and can be done in parallel. This is the point where the parallelism is introduced. Similarly, elements in the array X_i^k can also be cut into segments and the computations for different dual segment pairs can be done in parallel. For example, $X_1(0:3)$ and $X_1(8:11)$ form a dual segment pair, while $X_1(4:7)$ and $X_1(12:15)$ form another pair.

In general, if there are N elements in $X_0(k)$ (for simplicity, we consider the case $N = 2^{\mu}$, μ is an integer), we need μ steps to finish the computation. In each step, the array $X_i(k)$ is cut into segments and the computations for different dual segment pairs can be done in parallel. From Fig. 1, it is clear that to obtain a single dual pair in the final array $X_4(0:15)$, say $X_4(0)$ and $X_4(1)$, one single dual pair in X_3 , i.e. $X_3(0)$ and $X_3(1)$, is needed for the computation, which in turn requests another two dual pairs from X_2 , i.e. $X_2(0)$, $X_2(1)$, $X_2(2)$ and $X_2(3)$. If $X_0(0:15)$ is initially distributed over different nodes on a hypercube, the subsequent computations on a node depend on intermediate result of other nodes. It is this data dependence that brought up the need for inter-processor communication.

Suppose there are $p = 2^{\mu 1}$ nodes on a hypercube available for the computation, a most commonly used way to distribute the data array X_0

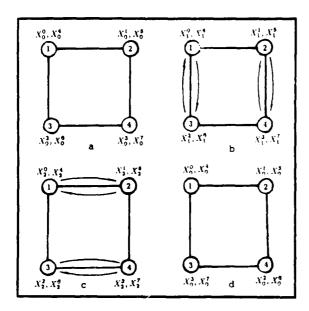


Figure 2: Data exchange

into different nodes is to divide X_0 sequentially into p segments of equal length n and then assign one segment to each node. In each computing step that requires communication, a data segment of length n must be exchanged between nodes for the next step computation. To reduce the amount of data transferred during the communication, we divide X₀ sequentially into 2p segments of equal length which form p dual segment pairs initially, then assign one dual segment pair to each node to start the computation. In each intermediate step that needs communication, every node sends half of its computed result(length n/2) to the nodes which need it for the next step computation, and wait for the information of same length from another node to arrive for continuing computation. Fig. 2 shows the information flow during the computation in the case of a hypercube with 4 nodes. Note in each step that needs communication, only half of the data stored in a node is exchanged with another node. X_0 is divided into 8 segments X_0^i , $i = 0, \dots, 7$. The circles in Fig.2 represent nodes, the letters outside the circles indicate the data segments stored in the local memory. At the first step, X_0^0 and X_0^4 are in the same dual segment pair, so are X_0^1 and X_0^5 and so on. Fig. 2a shows the initial data distribution of

 X_0^{i} 's. Each node starts work on the dual segment pair to obtain the updated value. Upon finishing step 1 computation, node 1 needs X_1^2 from node 3 to continue computation, while node 3 needs X_1^4 from node 1 for the next step. Thus, node 1 exchange information with node 3, as shown by the arc in Fig.2b, so does node 2 with node 4. Fig.2c shows the information flow at the last step. After that step, the computation in each node is completely independent because the distance between nodes in dual node pair is less than or equal to the data segment length, so all nodes have the necessary information available in their local memory for the continuing computation. In general, if there are $p = 2^{\mu 1}$ nodes on the hypercube and $N = 2^{\mu}$ elements in X_0 , (suppose $\mu > \mu 1$), μ steps of computations are needed for FFT and inter-processor communication is necessary only in the first μl steps. There is no communication at all for the remaining $\mu - \mu 1$ steps. Fig. 2d shows that if data segments were distributed into the nodes in another way, communication between non-adjacent nodes will occur (node I must communicate with node 4 and node 2 must communicate with node 3). To minimize communication cost and improve the algorithm performance, the following two points were considered when implementing the algorithm:

1. Eliminating fragmentary message passing and overlapping communication with computations. As was discussed previously, in the first μ_1 computation steps, each node needs to exchange half of its computed results with another node. If every single data element is exchanged one by one with the progress of the computation, communication cost will be overwhelmingly high due to the accumulation of start up cost. To overcome this difficult, each node sends all data items to be exchanged in one package after the computation in the current step has completed. While waiting for the data package from the other node to arrive, the node can start computing complex exponentiations $w^k = exp(2\pi ik/N)$ (including determining k by modular arithmetic and computing complex exponentiations) for the next step. When the data package from the other node arrives, these

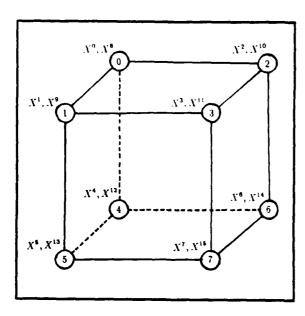


Figure 3: Data distribution

exponentiations can be used immediately in the multiplications and additions as described by formula (1).

2. Mapping all dual segment pairs that need mutual communication into the closest neighbors on a hypercube. The topological connection on a hypercube with $p=2^{\mu_1}$ makes each node connected directly with μ_1 other nodes (the closest neighbors). It is clear from the previous analysis that a specific node holding a dual segment pair needs to exchange intermediate results with μ_1 other dual segment pairs. These μ_1 dual segment pairs are mapped into the μ_1 closest neighbors of that specific node, thus all pairwise communications between dual segment pairs can be done directly without transferring through intermediate nodes. Fig. 3 gives the mapping of dual segment pairs into a hypercube with eight nodes.

The algorithm for each node can be described as follows:

```
{get \mu and \mu1 from the host }

{get initial dual segment pair

X_0^i and X_0^j from the host }

for k=1 to \mu_1 do

{compute X_k^i and X_k^j

from X_{k-1}^i and X_{k-1}^j }

{find next node n_p to communicate }
```

```
{ send out X_k^i or X_k^j }
{ compute complex exponentiations while waiting for data from n_p to arrive }
end for
for k = \mu 1 + 1, \mu do
{ compute X_k^i, X_k^j from X_{k-1}^i, X_{k-1}^j }
end for
{ send result X_\mu^i, X_\mu^j back to the host }
The programming language is Fortran and the communication subroutines are provided by the operating system.
```

Numerical Experiment and Analysis:

The algorithm was implemented on a 32-node iPSC/2 hypercube. Extensive numerical computations have been carried out to study the performance of the algorithm. The cube can be partitioned into subcubes. Both the number of data points N and the number of nodes p were adjusted as parameters during the computational experiment, with N ranging from 1024 to 262144 (μ from 10 to 18) and p ranging from 1 to 32 (μ 1 from 0 to 5). The algorithm has also been incorporated into a fast PDE solver to solve large scale reservoir simulation problems.

Fig. 4 gives the timing curve obtained from numerical computations for different μ values, where $N=2^{\mu}$ is the number of data elements to be transformed, $p=2^{\mu_1}$ is the number of nodes on the hypercube. The curves show that the computing time decreases rapidly as the number of nodes increases, until the optimal number of nodes has been reached. For example, when $N=2^{13}=8096$, the optimal number of p is $p=2^4=16$. After that, the extra communication cost incurred by increasing the number of nodes will outweigh the gain from the reduced numerical computation time, so the total executing time will increase. The optimal number of nodes increases with N, as indicated in Fig. 4.

For theoretical analysis, we used

$$t = \alpha + \beta \tau$$

as communication model,

$$T_{A} = 2fNlog_{2}N$$

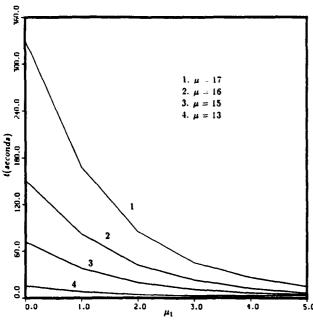


Figure 4: Timing curves

St. $\mu_1 = 1$ 1. $\mu_1 = 1$ 2. $\mu_1 = 2$ 3. $\mu_1 = 3$ 4. $\mu_1 = 4$ 5. $\mu_1 = 5$

Figure 5: Efficiency curves

as serial running time model on one node, and

$$T_p = 2f(\log_2 N)N/p + 10p(\alpha + \beta N/p)$$

$$+(\alpha+\beta N/p)log_2p$$

as parallel running time model on p nodes. a is the start-up time for communication, β is the time required to send one word to the closest neighbors and f is the time needed for one floating point multiplication. The value of these parameters are the same as used in [8]. There are three terms in the parallel running time model. The first term represents the numerical computations distributed to each node which is a decreasing function of p. The second term reflects the communication time for loading program, distributing data to and collecting the final results from all the nodes, this term increases linearly with the number of nodes p since broadcasting can not be used to distribute different data segments to all nodes. The last term gives the time spent on internodes communication during the computation. This term is usually much smaller than the second term since the communication takes place only between the closest neighbors in the first log₂p steps. The timing curves given by mathematical performance models are very close to the result obtained from the numerical computation. Those curves have been omitted to save the space.

The optimal number of nodes p_{opt} as a function of N can be obtained by solving the minimum of T_p . If the last communication term is omitted, the approximate solution will be:

$$p_{opt} = \sqrt{N log_2 N}/2$$

which indicates $p_{opt} \propto O(n^{1/2}(\log_2 N)^{1/2})$.

Fig. 5 gives the algorithm efficiency curves given by performance models. Different curves correspond to different number of nodes in the computation. The algorithm efficiency is defined as:

$$e = T_s/(T_p p)$$

Ideally, if there were no overhead and communication, e should be unity, meaning the computation will finish in 1/p the serial running time. The figure shows that there is a certain range of μ ($N=2^{\mu}$ is the number of data elements to be transformed) in which the efficiency increases very fast as μ increases. The algorithm is switching quickly from communication dominance to numerical computation dominance in

this range. For a 32 node iPSC/2 hypercube, the algorithm efficiency will be over 80 percent when N is greater than (2¹⁴). In summary, the algorithm introduced here is very efficient when implemented on a hypercube. High efficiency can be achieved by eliminating fragmentary message passing, overlapping computation and communication and taking advantage of rich connections available in the cube. The algorithm can also be used as a core routine in the time series analysis or spectral analysis algorithm on distributed memory computers to speed up the large scale computation.

Acknowledgment: The author appreciates very much the information about the available references provided by Professor D. Walker.

References

- [1] D. H. Bailey, A high-performance fast Fourier transform algorithm for Cray-2, J. of Supercomputing, 1 (1987), pp.43-60.
- [2] G. D. Bergland, A fast Fourier transform for real-valued series, Comm. ACM, 11 (1968), pp. 703-713.
- [3] W. L. Briggs, Multiprocessor FFT methods, SIAM J. Sci. Stati. Computing, 8 (1987), pp.s27-s42.
- [4] Brigham, The Fast Fourier Transform, Prentice-Hall, Englewood Cliffs, NJ, 1974.
- [5] W. T. Cochran et al., What is the fast Fourier transform? IEEE Trans. Audio. Electroacoustics, An-15 (1967), pp.45-55.
- [6] J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Comp., 19 (1965), pp. 297-301.
- [7] W. M. Gentleman and G. Sande, Fast Fourier transforms for fun and profit, 1966 Fall Joint Computer Conference, AFIPS Proc., 29 (1966), pp.563-578.

- [8] M. T. Heath et al., Parallel solution of triangular systems on distributed-memory multiprocessors, SIAM J. Sci. Statist. Comput. 9 (1988), pp.558-587.
- [9] R. W. Hockney, A fast direct solution of Poisson's equation by Fourier analysis, J. ACM, 12 (1965), pp.95-113.
- [10] D. G. Korn and J. J. Lambiotte, Computing the fast Fourier transform on a vector computer, Math. Comp. 33 (1979), pp. 977-992.
- [11] J. M. Ortega, Introduction to parallel and vector solution of linear systems, Plenum Press, New York, 1988.
- [12] Y. Saad and M. Schultz, Data communication in parallel architectures, dept. of comp. sci. rep. RR/461, Yale Univ., 1986.
- [13] P. N. Swarztrauber, FFT algorithm for vector computers, Parallel Computing, 1(1984), pp.45-63.
- [14] _____, Vectorizing the FFT's, in Parallel Computations, G. Rodrigue, ed., Academic Press, New York, 1982.
- [15] C. Temperton, Fast mixed-radix real Fourier transforms, J. Comput. Phys., 52 (1983), pp.340-350.
- [16] _____, Implementation of a self-sorting inplace prime factor FFT algorithm, J. Comput. Phys., 58 (1985), pp.283-299.
- [17] _____, A note on the prime factor FFT algorithm, J. Comput. Phys., 52 (1983), pp.198-204.
- [18] D. W. Walker, Portable programming within a message-passing model: the FFT as an example, Proceedings of the Third Conference on hypercube concurrent computers and applications, Ed. G. Fox, ACM Press, 1988.

Distributed Evaluation of an Iterative Function for All Object Pairs on an SIMD Hypercube

Fikret Ercal

Department of Computer Engineering and Information Sciences Bilkent University, Ankara, TURKEY

Abstract

An efficient distributed algorithm for evaluating an iterative function on all pairwise combinations of C objects on an SIMD hypercube is presented. The algorithm achieves uniform load distribution and minimal, completely local interprocessor communication.

1 Introduction

The problem addressed here is the following: Given a set of C objects uniformly distributed among the processors of an SIMD hypercube, and an operation on pairs of objects which may possibly modify the objects, is there a way to efficiently evaluate the operation iteratively on all the possible C(C-1)/2 pairwise combinations of the C objects in a distributed fashion? This problem arises for example in the context of parallel k—way graph partitioning on a hypercube [1], and in the scheduling of a roundrobin tournament between C players using C/2 courts, where the paths between courts form a hypercube interconnection. Matches between

players are to be scheduled so that the courts are maximally utilized and the players do minimal walking between courts.

In an earlier study [4], a distributed solution to the problem for an MIMD hypercube was presented, and shown to be optimal with respect to processor utilization and communication. In this paper, we solve the same problem for an SIMD hypercube. Two important constraints in the iterative application of the function make the otherwise trivial problem a non-trivial one : 1) the objects might get modified by the application of the operation, (i.e. not read-only) and 2) the result of the current step depends on the state of the objects after the previous step (iterative). Since the operation can change the objects, a consisteny problem arises if multiple copies of the same object exist simultaneously in the distributed system. Therefore, only one copy of an object must be allowed in the system.

The key to an efficient distributed pairwise combining algorithm is the appropriate scheduling of communication of the objects between the processors so that all possible pairs

meet exactly once, and no redundant computations occur. To achieve this, we require each processor to communicate with only its nearest neighbors, and do some usefull work after each communication. We present a fully distributed algorithm which maximally utilizes the system and uses minimal interprocessor communication. The algorithm comprises p + 1 phases, where p is the dimension of the hypercube. Each phase consists of two subphases - an object-circulation subphase, and a window-fragmentation subphase. Object-circulation subphase make use of the SIMD data circulation algorithm given in [2] with a simple modification to handle variable window sizes.

The paper is organized as follows: In section 2, we present a fully distributed algorithm using only local inter-processor communication for solving the pairwise-evaluation problem on an SIMD hypercube. In section 3, the algorithm is shown to be optimal. Section 4 concludes the paper with a brief discussion.

2 Distributed Pairwise-Evaluation on an SIMD Hypercube

We use the following notation in specifying the algorithm:

Given a processor numbered k, $0 \le k \le P-1$

 $b_d(k)$: d—th bit of the binary representation of k $N^d(k)$: the neighbor processor whose binary representation differs from k in only the d—th bit $C1_k$, $C2_k$: objects assigned to processor k $P = 2^p$: the number of hypercube processors $C = 2^c$: the total number of objects

Pairwise-Evaluation Algorithm listed below evaluates a given function for all C(C-1)/2 pairwise combinations of C objects using C/2 processors. Initially, each processor P_k contains two of the C objects, labeled $C1_k$ and $C2_k$, with no two processors containing the same object. The processors alternate between computation and communication, with each processor repeatedly performing: 1) a pairwise operation on the two locally held objects, and, 2) communication of one of the objects to a neighbor processor, in turn receiving some other object from a neighbor.

SIMD Distributed Pairwise-Evaluation Algorithm:

Processor P_k executes:

```
1. for d \leftarrow p to 0 do
     for s \leftarrow 1 to 2^d - 1 do
2.
           operate on the pair (C1_k, C2_k);
3.
           \operatorname{send}(C2_k, N^{h(d,s)}(k));
4.
           recv(C2_k, N^{h(d,s)}(k));
5.
6.
     endfor
7.
           operate on the pair (C1_k, C2_k);
8.
           if (d > 0) then
             if (b_{d-1}(k) = 1) then
9.
                \operatorname{send}(C1_k, N^{(d-1)}(k));
10.
                recv(C1_k, N^{(d-1)}(k));
11.
12.
             else
                \operatorname{send}(C2_k, N^{(d-1)}(k));
13.
                recv(C2_k, N^{(d-1)}(k));
14.
             endif
15.
16.
           endif
17. endfor
```

The key requirement is that the objects be moved between the processors in such a way

that each possible pair of objects comes together exactly once to enable the application of the pairwise operation on that pair. The algorithm has p+1 phases (indexed by "d"), where p is the number of dimensions of the hypercube. Each phase consists of two subphases - an object-circulation subphase where processors circulate their objects in closed windows (lines 2-6), and a window-fragmentation subphase where each window subdivides into two isolated windows (lines 8-16). The window structure thus changes from phase to phase, with 2^{p-d} independent windows of size 2^d being formed during phase d, as illustrated for a 4-dimensional hypercube in Fig. 1.

For an MIMD hypercube, object-circulation in a window of size 2^d can be easily done by repeatedly performing, $2^d - 1$ times, a circular shift of 1 among the processors belonging to the came window. On the other hand, due to the central control in an SIMD hypercube, optimal circulation requires a special exchange sequence X_d as described in [3]. This sequence is defined recursively as in the following:

$$X_1 = 0, X_d = X_{d-1}, d-1, X_{d-1} (d > 1)$$

For example, $X_3 = 0, 1, 0, 2, 0, 1, 0$. Using X_d sequence, object circulation in a window of size 2^d is achieved by first circulating data in windows of size 2^{d-1} in parallel using X_{d-1} sequence, then performing a data exchange across the two windows (along bit d-1), and finally circulating the exchanged data in the two windows again using X_{d-1} sequence. In the algorithm given above, the notation h(d,i) is

used to denote the i-th number in the sequence X_d , $1 \le i \le 2^d$. As an example, h(3,1) = 0, h(3,2) = 1, h(3,3) = 0, and h(3,4) = 2.

During a phase, corresponding to one iteration of the d-loop of the algorithm, each processor keeps one of its objects (C1) local, while it repeatedly receives, transforms and passes on the second object (C2). Considering phase p, with all processors communicating in one single window, at the end of the $2^p - 1$ steps in the first part (the object-circulation subphase) of the phase, all objects constituting the various $C1_k$'s (denoted CS1) would have been matched up with respect to every object in the CS2 set (and the pairwise operation performed on each such generated pair). Thus the only pairings between objects that have not yet been formed are between the members of the CS1 set and likewise, mutually among the members of CS2. The window-fragmentation subphase of phase p involves pairwise communication exchanges between each processor and its neighbor whose address differs in the highest bit. During this subphase, each processor P_k with highest address bit of one $(b_{p-1}(k)=1)$, swaps its C1 object for the C2 object of its partner processor(P_l , with $b_{p-1}(l)=0$). Thus, after this communication subphase, all processors P_k with $(b_{p-1}(k)=1)$, will only have objects from the original CS2 set, while all processors with $(b_{p-1}(k)=0)$ will have all the objects comprising the original CS1 set. This subphase is labeled the "window-fragmentation subphase" because the window gets fragmented into two smaller windows and no communica-

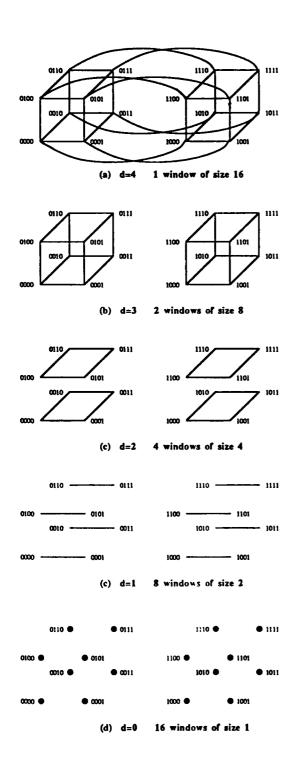


Figure 1: Illustration of window formation in different phases of the Distributed PC algorithm

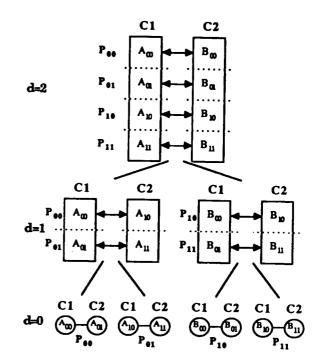


Figure 2: Illustration of Distributed PC algorithm on a 2-D hypercube (4 processors)

tion takes place thereafter between the processors in the "highest-bit-1" window and those in the "highest-bit-0" window. Thus in phase (p-1), two windows of size 2^{p-1} are formed for the object-circulation subphase and communication occurs between processors differing in their (p-2)th bit during the window-fragmentation subphase.

During each phase of the algorithm, new object-pairs meet at the processors, for application of the pairwise operation. The algorithm guarantees that during an outer pass, no pair of objects is ever matched up more than once. Fig. 2 is used to illustrate this "no-repetition" property of the algo-

the window-fragmentation subphase, the effects of the alternating object-circulation subphase are intentionally omitted. Eight objects are shown, mapped onto four processors, two objects per processor. During phase 2 (d = 2), the application of the objectcirculation subphase results in the generation of all possible pairwise combinations with one object from CS1 $(A_{00}, A_{01}, A_{10}, A_{11})$ and the other from CS2 $(B_{00}, B_{01}, B_{10}, B_{11})$. Ignoring for now the actual permutation of the C2 objects that will result at the end of the objectcirculation subphase, and assuming it to be as shown, the window-fragmentation subphase of phase 2 will result in the state shown for d = 1. Processors P_{00} and P_{01} are left with objects $A_{00}, A_{01}, A_{10}, A_{11}$, whereas P_{10} and P_{11} now have objects $B_{00}, B_{01}, B_{10}, B_{11}$. After the window-fragmentation phase of phase 2, P_{0*} and P_{1*} do not ever again communicate with each other. Since no pairwise combinations involving two A-objects had occurred during phase 2, and since none of the B-objects can any longer meet any of the A-objects, all pairs of objects that align at any processor are unique combinations that have not occurred earlier. The same property clearly holds recursively, as illustrated in the figure.

In the next section, we formally prove the correctness of the distributed algorithm.

3 Proof of Optimality

Lemma 1 The total number of pairwise object

In order to focus on the nature of combinations that occur during execution of the dow-fragmentation subphase, the ef-algorithm is C(C-1)/2.

Proof: Each processor performs one pairwise comparison during every step of every phase of the algorithm, as is clear from the algorithm specification. The number of steps in phase d is 2^d . Hence the total number of pairwise combinations tried is:

$$2^{p} * \sum_{d=p}^{0} 2^{d} = 2^{p} * (2^{(p+1)} - 1)$$

= $P(2P - 1) = C(C - 1)/2$

Lemma 2 Given any objects C_i and C_j , the combination (C_i, C_j) can occur at most once during execution of the algorithm.

Proof: Let d be the earliest phase that the combination (C_i, C_j) occurs. most one such match can occur during the object-circulation subphase of phase d. For such a match to occur, one of them must belong to the C1-object-set and the other to the C2-object-set. Since they belong to different object-sets, during the window-fragmentation subphase of phase d, C_i and C_j will necessarily end up in processors P_k , P_l , where 1 and kdiffer at least in bit d-1, and hence P_k and P_l belong to different windows. Obviously, they cannot get matched in any later phase d' < d. Hence at most one match (C_i, C_j) can occur during an outer pass.

Theorem 1 Given any two objects C_i and C_j , load distribution and minimal, completely local the pairwise combination (C_i,C_j) occurs exactly once during execution of the algorithm.

Proof: Theorem 1 follows immediately from lemma 1 and lemma 2. By lemma 1, a total of C(C-1)/2 pairwise combinations occur, and by lemma 2, no combination (C_i, C_j) can occur more than once. Since the number of possible distinct combinations of object pairs is C(C -1)/2, all possible matches must occur exactly once during execution of the algorithm.

Theorem 1 implies that as regards to computation, the algorithm is optimal since every processor is busy during each computational step and no duplicate computations occur. With respect to communication too, under the constraint of computational load balancing and uniform data distribution, each processor can only contain two objects, and after performing the pairwise operation on its currently held pair, it will have to send out at least one object and receive one object in order to perform useful computation at the next step. The algorithm causes only one object to be sent and one object received by each processor at each step, i.e., the algorithm performs minimal communication.

Discussion 4

An efficient distributed algorithm for evaluating an iterative function on all pairwise combinations of C objects on an SIMD hypercube is presented, and it is shown to achieve uniform

inter-processor communication.

In case that C > 2P, the algorithm can be extended in a straightforward fashion. For C = MP, M = 2k, k > 1, groups of M/2 objects should be considered in place of single objects in the presented algorithm. Now, instead of a single pairwise operation, $(M/2)^2$ pairwise operations are performed at each step of the algorithm between member partitions of the two (M/2)-ary object-groups in a processor. With such a (M/2) – ary group of objects in place of single objects, the algorithm for distributed PC is essentially the same as above, except for an additional set of operations between the components of each (M/2) – ary group of objects.

References

- [1] P. Sadayappan, F. Ercal and J. Ramanujam, "Parallel Graph Partitioning on a Hypercube," Proc. of Fourth Conf. on Hypercube Concurrent Computers and Applications, March, 1989.
- [2] S.Ranka and S. Sahni, Hypercube Algorithms For Image Processing and Pattern Recognition, Bilkent University Lecture Notes, Springer-Verlag, in press.
- [3] E. Dekel, D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms," SIAM Journal on Computing, 1981, pp. 657-675.
- [4] P. Sadayappan, F. Ercal and J. Ramanujam, "Distributed Generation of Pairwise Combinations on a Hypercube," in Proceedings of Parallel Computing 89, Leiden, The Netherlands, August 1989.

The Complexity of Reshaping Arrays on Boolean Cubea

S. Lennart Johnsson*
Department of Computer Science
Yale University
New Haven, CT 06520
Johnsson@cs.yale.edu, Johnsson@think.com

Ching-Tien Ho[†]
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
Ho@ibm.com

Abstract

Reshaping of arrays is a convenient programming primitive. For arrays encoded in a binary-reflected Gray code reshaping implies code change. We show that an axis splitting, or combining of two axes, requires communication in exactly one dimension, and that for multiple axes splittings the exchanges in the different dimensions can be ordered arbitrarily. The number of element transfers in sequence is independent of the number of dimensions requiring communication for large local data sets, and concurrent communication. The lower bound for the number of element transfers in sequence is $\frac{K}{2}$ with Kelements per processor. We present algorithms that is of this complexity for some cases, and of complexity K in the worst case. Conversion between binary code and binary-reflected Gray code is a special case of reshaping.

1 Introduction

In computer systems locality of reference has had a significant impact on performance ever since memory hierarchies were introduced. In modern computer systems small memories in MOS technologies may be designed for higher speeds than larger memories. In multiprocessor systems with processors at d memory modules interconnected via a network, the access time for non-local information is typically considerably longer than local access. Moreover, the access time depends upon the network topology, congestion and bandwidth of the communications network. The reference pattern has a significant impact on the optimal data allocation in networks that have a non-uniform distance between pairs of nodes, such as Boolean cube networks.

In well structured computations the data is conveniently represented by arrays. Many algorithms require local references in a Cartesian space corresponding to

the array. Explicit methods for the solution of partial differential equations are examples thereof. Preserving the locality in the Cartesian space when mapped to the processor network is important with respect to performance. The binary-reflected Gray code is often used to accomplish this task in Boolean cube networks. Successive integers in the decimal encoding differ by one bit in their Gray code encoding. This property is used in CM-Fortran [1], Thinking Machines Corp. version of Fortran 8X [11] for the Connection Machine. In this language implementation, array axes are by default encoded in a binary-reflected Gray code.

Some important algorithms with a regular communication pattern depend on local references in a Boolean space. For instance, the Fast Fourier Transform requires communication in the form of a butterfly network, which implies communication between adjacent nodes in a Boolean space with corresponding nodes in different ranks mapped to the same processor. In many scientific and engineering applications algorithms that depend upon both types of access patterns may be used, and conversion between the two storage forms may be important.

Many recursive algorithms make use of axis splitting, or combining. An example is the data parallel implementation [2] of the divide-and-conquer algorithm by Dongarra and Sorensen [3] for computing eigenvalues of symmetric tridiagonal systems. Array manipulation through operations such as RESHAPE in Fortran 8X and APL, impacts the encoding for binary-reflected Gray coded axes. The encoding of binary coded axes is unaffected.

Different axes may have different encoding. For instance, if butterfly computations are performed along one axis, and nearest-neighbor communications in a Cartesian space along the other axis of a two-dimensional array, then binary encoding of the first axis and binary-reflected Gray code encoding of the second axis is desirable. Furthermore, the encoding of a single axis may be mixed. Typically the number of array elements along an axis exceeds the number of processors allocated to the axis, forcing several elements along an axis to be allocated to the memory of each proces-

^{*}The author is also with Thinking Machines Corp., 245 First Street, Cambridge, MA 02142. This work was supported in part by AFOSR-89-0382 and ONR Contract No. N00014-86-K-0310.

[†]Part of the work was done while the author was with the Department of Computer Science, Yale University.

sor with the array elements being allocated as evenly as possible. Cyclic and consecutive [6] allocation are two common schemes for assigning multiple elements to processors. With local random access memories distance is not an issue in determining the encoding for the local memories. Binary encoding is typically used for the local part of an axis, and binary-reflected Gray code for the processor part.

As an example consider a two-dimensional logic array A of shape $P \times Q$ allocated to an $N_1 \times N_0$ physical array of processors, where $P = 2^p$, $Q = 2^q$, $N_1 = 2^{n_1}$, $N_0 = 2^{n_0}$, $p \ge n_1$ and $q \ge n_0$. The data allocation is consecutive, and each array axis is encoded in a binary-reflected Gray code. Bit m in the address space is denoted g_m if encoded in a binary-reflected Gray code, and b_m if encoded in binary code. Bit zero, or dimension zero, is the least significant, and the rightmost dimension in our expressions. The symbol || denotes concatenation of two fields. Axes are also labeled right to left. We illustrate the allocation as follows

$$\underbrace{ g_{p-1}^1 g_{p-2}^1 \cdots g_{p-n_1}^1}_{\text{paddr}^1} \underbrace{ g_{p-n_1-1}^1 g_{p-n_1-2}^1 \cdots g_0^1}_{\text{maddr}^1} || \underbrace{ g_{q-1}^0 g_{q-2}^0 \cdots g_{q-n_0}^0}_{\text{paddr}^0} \underbrace{ g_{q-n_0-1}^0 g_{q-n_0-2}^0 \cdots g_0^0}_{\text{maddr}^0}).$$

The processor address for an element (i,j) of the logic array is formed as $(paddr^1(i)||paddr^0(j))$, and the local storage address is $(maddr^1(i)||maddr^0(j))$, where $G_p(i) = (g_{p-1}^1 g_{p-2}^1 \cdots g_0^1)$ is the binary-reflected Gray code encoding of i, and $G_q(j) = (g_{q-1}^0 g_{q-2}^0 \cdots g_0^0)$ is the binary-reflected Gray code encoding of j. Reshaping the logic array into a one-dimensional array such that $(i,j) \rightarrow iQ + j$ preserving the assignment of bits in the logic array to bits in the physical address space implies a code conversion for axis zero if i is odd, and data motion within n_0 dimensional subcubes. The result is an allocation of the form

$$\underbrace{\left(\underbrace{g_{p+q-1}g_{p+q-2}\cdots g_{p+q-n_1}}_{\text{paddr}^1}\underbrace{g_{p+q-n_1-1}g_{p+q-n_1-2}\cdots g_q}_{\text{maddr}^1}\right)}_{\text{paddr}^0}$$

where, as shown later, $g_{m+q} = g_m^1$, $m \in \{0, \dots, p-1\}$ and $g_m = g_m^0$, $m \in \{0, \dots, q-2\}$. The value of g_{q-1} depends upon the value of g_q . Figure 1 illustrates the data motion.

Note that whereas the initial data allocation was consecutive the data allocation after reshaping is not. If a consistent data allocation is desired, i.e., the same data allocation scheme before and after reshaping, then it is in general necessary to change the assignment of dimensions in the logic address space to dimensions in the physical address space. A dimension permutation

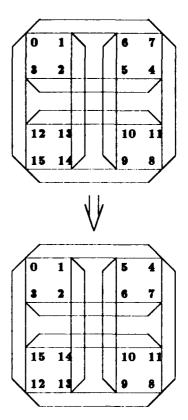


Figure 1: Reshaping an 1×16 array to a 4×4 array.

[4,13,12,15,10,5] in the form of an n_0 step right cyclic shift, or $p-n_1$ steps left cyclic shift on the dimensions in the field (maddr¹||paddr⁰) is required, in combination with code conversion.

With consecutive allocation of A and a binary encoding of local addresses, and a binary-reflected Gray code encoding of processor addresses, the processor address of element (i, j) is formed by computing the address from the binary-reflected Gray codes of $\lfloor i/N_1 \rfloor$ and $\lfloor j/N_0 \rfloor$. The local memory address is determined from the binary codes of $i \mod N_1$ and $j \mod N_0$. The encoding of the address field is

$$\underbrace{(g_{p-1}^1g_{p-2}^1\cdots g_{p-n_1}^1}_{\text{paddr}^1}\underbrace{b_{p-n_1-1}^1b_{p-n_1-2}^1\cdots b_0^1}_{\text{maddr}^1}||}_{\text{paddr}^0}$$

$$\underbrace{g_{q-1}^0g_{q-2}^0\cdots g_{q-n_0}^0}_{\text{paddr}^0}\underbrace{b_{q-n_0-1}^0b_{q-n_0-2}^0\cdots b_0^0}_{\text{maddr}^0}).$$

Reconfiguration of the processor array is equivalent to changing the assignment of dimensions in the logic address space to dimensions in the physical address space. A dimension permutation is required. If the encoding of the local address field is different from the processor address field, then a code conversion is required in combination with the dimension permutation. Reconfiguration of a processor array may be required to assure

that all operands use the same physical machine configuration, as for instance in matrix multiplication on the Connection Machine [8]. The Connection Machine Fortran compiler allocates logic arrays to the processors by defining a processor array congruent to the logic array for each array. Hence, in the matrix multiplication $C \leftarrow A \times B$ all three matrices may assume a different shape of the processor array.

In this paper, we show how an axis splitting, or the combining of two axes into one, can be performed by a single exchange operation. For multiple axes split/merge operations, the number of element transfers in sequence is independent of the number of axes created or merged, if the communication system allows concurrent communication in all required dimensions. The number of element transfers in sequence is only a function of the size of the local data set, if there is a large local data set. The minimum number of element transfers in sequence is equal to the number of dimensions requiring communication. The conversion between binary-reflected Gray code and binary code is equivilent to reshaping between a one-dimensional array and a $2 \times 2 \times \cdots \times 2$ array of dimension n.

The algorithms we give for reshaping and code conversion are either asymptotically optimal, or optimal within a factor of two with respect to data transfer time. The control information can be computed locally from the node address. The code conversion can start in any dimension, and the required exchanges can be carried out in dimensions ordered arbitrarily. This property allows reshaping by concurrent communication in all required dimensions, if the size of the local data set exceeds the number of dimensions requiring communication. Compared to the algorithms in [6,7] the new algorithms avoid the pipeline delay. Here we only treat the case with an entire axis encoded in either binary code, or binaryreflected Gray code. Furthermore, we assume a fixed assignment of dimensions in the logic address space to dimensions in the physical address space. Reshaping combined with dimension permutations is considered in

The paper is organized as follows. Notation and definitions are introduced next. Array reshaping is discussed in Section 3. The conversion between binary-reflected Gray code and binary code is discussed in Section 4, followed by summary in Section 5.

2 Preliminaries

A Boolean n-cube has $N=2^n$ nodes. Two nodes are adjacent if and only if their addresses differ in exactly one bit. The binary encoding of i is $B_n(i) = (b_{n-1}b_{n-2}\cdots b_0)$ and its binary-reflected Gray code encoding is $G_n(i) = (g_{n-1}g_{n-2}\cdots g_0)$. $Z_N = \{0,1,\cdots,N-1\}$ and (1^j) is a string of j instances of

a bit with value one. " $\{|"\}$ " is the concatenation symbol. For the complexity estimates we assume bi-directional channels and concurrent communication on all channels. The number of elements per node is K. \hat{G}_n is the sequence of n-bit binary-reflected Gray codes for Z_N , i.e., $\hat{G}_n = (G_n(0), G_n(1), \dots, G_n(2^n - 1))$.

Definition 1 [14] The binary-reflected Gray code is defined recursively as follows.

$$\hat{G}_1 = (G_1(0), G_1(1)), \text{ where } G_1(0) = 0, G_1(1) = 1.$$

$$\hat{G}_{n+1} = \begin{pmatrix} 0 || G_n(0) \\ 0 || G_n(1) \\ \vdots \\ 0 || G_n(2^n - 2) \\ 0 || G_n(2^n - 1) \\ 1 || G_n(2^n - 1) \\ 1 || G_n(2^n - 2) \\ \vdots \\ 1 || G_n(1) \\ 1 || G_n(0) \end{pmatrix}$$

In the following we always refer to the binary-reflected Gray code defined above.

Corollary 1 The highest order bit is the same in the binary code and the binary-reflected Gray code. The remaining bits in the encoding of $i \in \mathbb{Z}_{N/2}$ are defined by $G_{n-1}((b_{n-2}b_{n-3}\cdots b_0))$. The remaining bits in the encoding of $i \in \mathbb{Z}_N - \mathbb{Z}_{N/2}$ are defined by $G_{n-1}((\bar{b}_{n-2}\bar{b}_{n-3}\cdots \bar{b}_0))$. Thus,

$$G_n((b_{n-1}b_{n-2}\cdots b_0)) = \begin{cases} b_{n-1}||G_{n-1}((b_{n-2}b_{n-3}\cdots b_0)), \\ if b_{n-1} = 0, \\ b_{n-1}||G_{n-1}((\bar{b}_{n-2}\bar{b}_{n-3}\cdots \bar{b}_0)), \\ if b_{n-1} = 1. \end{cases}$$

Proof: From Definition 1.

Corollary 2 The integer encoded in the neighbor of node $G_n(i)$ in cube dimension j is $G_n(i \oplus (1^{j+1}))$, i.e., $G_n(i) \oplus 2^j = G_n(i \oplus (1^{j+1}))$.

Proof: It follows from Corollary 1.

Definition 2 With binary-reflected Gray code encoding of an N-element one-dimensional array A[i], $i \in Z_N$ into an n-cube, address $G_n(i)$ contains A[i].

Lemma 1 [14] $b_m = g_{n-1} \oplus g_{n-2} \oplus \cdots \oplus g_m$, $m \in \mathbb{Z}_n$. Conversely, $g_m = b_m \oplus b_{m+1}$, $m \in \mathbb{Z}_n$ with $b_n = 0$.

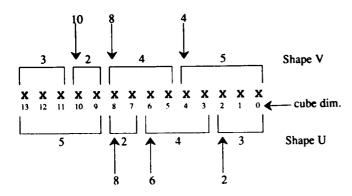


Figure 2: Reshaping between two arrays with binary-reflected Gray code encoded on a Boolean cube.

Definition 3 Let A be an array of shape $U_{d-1} \times U_{d-2} \times \cdots \times U_0$, $U = (U_{d-1}, U_{d-2}, \cdots, U_0)$, $U_m = 2^{u_m}$, $m \in \mathcal{Z}_d$, $V = (V_{d'-1}, V_{d'-2}, \cdots, V_0)$, $V_m = 2^{v_m}$, $m \in \mathcal{Z}_{d'}$ and $\prod_{m=0}^{d-1} U_m = \prod_{m=0}^{d'-1} V_m$. The reshape function $\rho(U, V)$ transforms the shape of the array A from U to V.

Let $\tilde{u}_k = (\sum_{m=0}^k u_m) - 1$, $\tilde{v}_k = (\sum_{m=0}^k v_{r,k}) - 1$, $\tilde{\mathcal{U}} =$ $\{\tilde{u}_k \mid 0 \le k < d-1\}, \, \tilde{\mathcal{V}} = \{\tilde{v}_k \mid 0 \le k < d'-1\} \text{ and } \mathcal{D} = \{\tilde{v}_k \mid 0 \le k < d'-1\}$ $(\tilde{\mathcal{U}} \cup \tilde{\mathcal{V}}) - (\tilde{\mathcal{U}} \cap \tilde{\mathcal{V}})$. The sets $\tilde{\mathcal{U}}$ and $\tilde{\mathcal{V}}$ are the sets of most significant dimensions for the axes of the shapes U and V, with the most significant axes excluded. For instance, if $U = (2^5, 2^2, 2^4, 2^3)$ and $V = (2^3, 2^2, 2^4, 2^5)$, then $\tilde{\mathcal{U}} =$ $\{8,6,2\}, \ \tilde{\mathcal{V}} = \{10,8,4\} \text{ and } \mathcal{D} = \{10,6,4,2\}, \text{ Figure 2}.$ To form the shape V from U communication is required in the set of dimensions defined by $\tilde{\mathcal{U}} - \tilde{\mathcal{V}}$ for axes being combined into one, and the set of dimensions defined by $\tilde{\mathcal{V}} - \tilde{\mathcal{U}}$ for axes being split. \mathcal{D} is the set of dimensions for which communication is required for changing the shape U into V. \mathcal{D}_{paddr} is the subset of dimensions in \mathcal{D} assigned to processor dimensions in the physical address space. $\mathcal{D}_{maddr} = \mathcal{D} - \mathcal{D}_{paddr}$ is the set of dimensions in D assigned to local memory dimensions in the physical address space.

3 Reshaping Arrays

Lemma 2 below states the fact that splitting an axis into two, or merging two axes into one, requires a code change in precisely one dimension.

Lemma 2 Assume node $G_n(i)$ contains element A[i], $i \in \mathcal{Z}_N$, initially. If all nodes $i = (b_{n-1}b_{n-2}\cdots b_0)$ such that $b_m = 1$ exchange data in dimension m-1 for any $m \in \{1, 2, \dots, n-1\}$, then node $G_{n-m}((b_{n-1}b_{n-2}\cdots b_m))||G_m((b_{m-1}b_{m-2}\cdots b_0))$ contains element A[i] after the exchange.

Proof: Assume that the reshape operation is $U = (2^n) \rightarrow V = (2^{n-m}, 2^m)$, and that address $G_n(i) =$

From the binary encoding $b_j(k) = b_{m+j}(i)$, $j \in \mathbb{Z}_{n-m}$, and $b_j(\ell) = b_j(i)$, $j \in \mathbb{Z}_m$. By Lemma 1, $g_j(k) = b_j(k) \oplus b_{j+1}(k) = b_{m+j}(i) \oplus b_{m+j+1}(i) = g_{m+j}(i)$, for all $j \in \mathbb{Z}_{n-m}$, and $g_j(\ell) = b_j(\ell) \oplus b_{j+1}(\ell) = b_j(i) \oplus b_{j+1}(i) = g_j(i)$ for all $j \in \mathbb{Z}_{m-1}$. But, $g_{m-1}(\ell) = b_{m-1}(\ell) \oplus b_m(\ell) = b_{m-1}(i)$ and $g_{m-1}(i) = b_{m-1}(i) \oplus b_m(i)$, i.e.,

$$g_{m-1}(\ell) = \begin{cases} g_{m-1}(i), & \text{if } b_m(i) = 0, \\ \hline g_{m-1}(i), & \text{if } b_m(i) = 1. \end{cases}$$

Hence, if $b_m(i) = 0$ then $G_n(i) = G_{n-m}(k)||G_m(\ell)$ and no data motion is necessary for reshaping. But, if $b_m(i) = 1$ then an exchange is required in dimension m-1, and only in dimension m-1, since this dimension is the only dimension in which the code for i and (k, ℓ) differs.

The change in the binary-reflected Gray code caused by an axis splitting, or the merging of two axes, is limited to the most significant dimension of the lower ordered axis in the created pair of axes. The pairs of addresses exchanging content in a given dimension depend upon the order of exchanges in the case of multiple axes splittings. The control of the exchange is derived from b_m in the encoding of i. The index i assigned to an address changes if a more significant controlling dimension is one. For example, consider the reshaping of an array of 8 elements encoded in a binary-reflected Gray code to an array of $2 \times 2 \times 2$ elements (which is equivalent to conversion to binary code). Figure 3 shows exchanged data in boldface, and two exchange orders: dimension one then zero, or zero then one. As is apparent from Figure 3, an exchange is carried out in dimension one between addresses 110 and 111 if the dimensions are treated in the order one first then zero, but not if the order is dimension zero first, then dimension one.

The current value of b_m that is assigned to a given address $(g_{n-1}g_{n-2}\cdots g_0)$ is easily determined from the address.

Lemma 3 If the number of exchanges in dimensions more significant than m is even, then the current value of logic dimension m assigned to an address $G_n(i) = (g_{n-1}g_{n-2}\cdots g_0)$ is b_m , otherwise it is \bar{b}_m .

The lemma follows directly from Corollary 2.

Half of the total number of elements need to be exchanged for any split/merge operation. Hence, the number of exchanges in which an element participates falls in

	ay code ignment	Ex		Ex din		Ex	-	Ex	
1	paddr	b ₂	- i	$\overline{b_1}$	i	<i>b</i> ₁	1	b ₂	i
ō	000	Ō	Õ	0	Ō	0	Ō	Ō	ō
1	001	0	1	0	1	0	1	0	1
2	011	0	2	1	3	1	3	0	3
3	010	0	3	1	2	1	2	0	2
4	110	1	7	1	6	0	4	1	8
5	111	1	6	1	7	0	5	1	7
6	101	1	5	0	5	1	7	1	5
7	100	1	4	0	4	1	6	1	4

Figure 3: Reshaping an array of 8 elements into a $2 \times 2 \times 2$ array.

the range $0 - |\mathcal{D}|$, depending upon its binary encoding. The total number of element exchanges is $|\mathcal{D}|^{\prod_{m=0}^{d-1} U_m}$ for changing shape U to shape V. We will now determine the number of element exchanges in sequence when the logic array is allocated to an n-cube, with $\prod_{m=0}^{d-1} U_m = K$ elements per processor.

Theorem 1 A lower bound for the number of element transfers in sequence for array reshaping affecting the encoding of processor dimensions is K/2 with K elements per processor.

Proof: Pick a dimension $d \in \mathcal{D}_{paddr}$. There are N/2 processors that need to transfer data across dimension d. There are K elements in each processor, and all elements need to be exchanged. The available bandwidth per dimension is N.

In the following, let $\delta = |\mathcal{D}_{paddr}|$.

Theorem 2 Changing the shape U to shape V preserving the assignment of logic dimensions to physical dimensions requires at most $\delta \lceil \frac{K}{\delta} \rceil$ element transfers in sequence with concurrent communication.

Proof: Let $\mathcal{D}_{paddr} = \{d_{\delta-1}, d_{\delta-2}, \cdots, d_0\}$. Partition the local data set of size K into δ sets of size at most $\lceil \frac{K}{\delta} \rceil$ each. Label the data sets from 0 to $\delta-1$. Each such set is assigned a sequence of dimensions including all dimensions in \mathcal{D}_{paddr} once. Different sets are assigned different sequences such that no two sets have the same first, second, third, etc., dimension. For instance, let data in set m be assigned the sequence of dimensions $d_m, d_{(m+1) \mod \delta}, \cdots, d_{(m-1) \mod \delta}$.

The upper bound in Theorem 2 differs from the lower bound by a factor of two. The upper bound can be improved in some cases. We give upper bounds that are almost identical to the lower bounds for two cases. Theorem 3 Changing the shape U to shape V preserving the assignment of logic dimensions to physical dimensions requires at most $\delta\lceil\frac{K}{2\delta}\rceil+1$ element transfers in sequence with concurrent communication, if no two elements of \mathcal{D}_{paddr} differ by one and $K>2\delta$.

Proof: Consider the merging of a single pair of axes, or splitting of an axis. Assume the communication occurs in dimension m-1. Consider a 2-cube formed by dimensions m and m-1. Label the four nodes according to $b_m b_{m-1}$. By Lemma 2, communication is only required between nodes 10 and 11. There exist two edge-disjoint paths between these two nodes of lengths one and three, respectively. By assigning $\lceil \frac{K}{2\delta} \rceil + 1$ elements to the path of length one and the remaining elements to the path of length three, $(\lceil \frac{K}{2\delta} \rceil + 1)$ element transfers in sequence are required.

If no two elements in \mathcal{D}_{paddr} differ by one, then the 2-cubes used for different data sets are disjoint. Thus, $\delta(\lceil \frac{K}{2\delta} \rceil + 1)$ element transfers in sequence are required. To reduce the communication complexity to $\delta \lceil \frac{K}{2\delta} \rceil + 1$, we slightly overlap the communications on the successive 2-cubes of a given data set. Without this overlap no data is sent along the length three path during the last two cycles of the routing of a data set. By sending two elements that have been routed with respect to the first 2-cube to the length-three path of the second 2-cube during the last two cycles of the routing phase of the first 2-cube (with one cycle each), the communication delay due to the length-three path is only paid once. Sending elements along the length-three path during the last two cycles of the first 2-cube will not interfere with the communication of the data set exchanged in the second 2-cube. The reduced complexity is valid if $\left\lceil \frac{K}{K} \right\rceil > 2$, i.e., some data set has at least three elements.

In the routing used for the proof of the bound, the number of elements routed along the length-one path and the length-three path differ by two only for the first 2-cube. For subsequent 2-cubes, the same number of elements are routed along each path, with the length-three path starting two cycles earlier. The first element on both paths arrives at the same time within the 2-cube except for the first 2-cube. If 2δ divides K and $K > 2\delta$, then the complexity is $\frac{K}{2} + 1$, which is only one element transfer above the lower bound. For $K \leq 2\delta$, there is no advantage of using the length-three paths over the algorithm used in the proof of Theorem 2.

If the reshape operation requires communication in dimensions m-1 and m (by creating an axis of length 2 encoded in dimension m), then dimension m cannot be used for rerouting to access unused communication links in dimension m-1. Unused links in dimensions lower than m-1 cannot be used either, since they do not connect to processors with unused links in dimension m-1. However, the following observation can be used to reduce the number of element transfers in sequence.

Lemma 4 For a reshape operation requiring communication in dimension m-1 none of the links in dimension m-1 is used in m-1 dimensional subcubes obtained through complementing any of the address dimensions that are more significant than m-1.

Proof: We need to show that in any m-1 dimensional subcube defined by dimensions m and higher, $b_m = 0$ if the address defining the subcube is obtained by complementing a single dimension of significance m or higher. But, by Lemma 1 complementing a single dimension g_j , $j \in \{m, m+1, \cdots, n-1\}$ complements b_m .

By using a pipelined algorithm instead of the nonpipelined maximally concurrent algorithm used for the upper bound in Theorem 3, the properties in Lemma 4 can be exploited to establish the following bound.

Theorem 4 Changing the shape U to shape V requires at most $\lceil \frac{K}{2} \rceil + 2\delta - 1$ element transfers in sequence, if for each dimension requiring communication there exists one more significant dimension not requiring communication and $K \geq 2\delta$.

Proof: The problem is equivalent to sending K elements along a path of length δ and each edge on the path is paired with a length-three path, disjoint with all other edges. If δ is even two edge-disjoint paths of length 2δ can be defined by combining length-three and length-one paths for different dimensions. If δ is odd, then two paths of length $2\delta - 1$ and $2\delta + 1$ can be defined in a similar way.

Several routing schemes yield the same complexity as the scheme used in the proof. For instance, by creating one path of length δ and one of length 3δ , and routing $\lceil \frac{K}{2} \rceil + \delta$ elements along the short route and $\lfloor \frac{K}{2} \rfloor - \delta$ elements along the long route the same routing time is achieved if $K \geq 2\delta$. For $K < 2\delta$, the latter approach degenerates to using a single path of length δ and the required time is $K + \delta - 1$, which is lower than if two paths of the same length were used. However, if $K < 2\delta$ then the time for reshaping by pipelining along one path is higher than, or at best the same as if the concurrent exchange algorithm in the proof of Theorem 2 is used.

Lemma 4 cannot be exploited directly for concurrent exchange sequences because an exchange in one dimension affects the set of edges being used in a subcube. This property follows from Lemma 3. For instance, if a 1×16 array is reshaped into a $4 \times 2 \times 2$ array, then if an exchange in dimension one is performed first the required exchanges in dimension zero are all on corresponding links in different subcubes instead of complimentary links.

4 Conversion between Gray code and binary code

Theorem 5 The conversion between a binary-reflected Gray code and binary code in either direction requires communication in n-1 dimensions, and at most $(n-1)\lceil \frac{K}{n-1} \rceil$ element transfers in sequence.

Theorem 5 follows from Theorem 2 and the observation that conversion from binary-reflected Gray code to binary code in an n-cube is equivalent to reshaping a one-dimensional array of size 2^n to an n-dimensional array of shape $2 \times 2 \times \cdots \times 2$.

In any algorithm according to Lemma 2 and Theorem 5 only half of the communications links in each of the n-1 dimensions are used in every step of the algorithm. Every path is of minimum length, and all minimum length paths are used evenly. The load on the communications network is minimal.

Conjecture 1 For the conversion between binary-reflected Gray code and binary code encodings of K elements per processor in an n-cube, a lower bound is $K \frac{n-1}{n}$.

For n = 2, the conjecture follows from Theorem 3. For n > 2 only the most significant dimension requires no communication.

Corollary 3 The conversion between binary-reflected Gray code and binary code encoding in an n-cube can be performed as an arbitrary sequence of communications in dimensions: $\{0, 1, \dots, n-2\}$.

The corollary follows from the observation that the control is completely determined by the binary encoding of i.

An algorithm proceeding from dimension n-2 to dimension 0 is depicted in Figure 4. Initially, processor $G_4(i)$ contains data of index i. After the conversion, i is assigned to processor $B_4(i)$. The algorithm is described below. Several other algorithms are given in [7].

/* Converting Gray code to binary code starting from the most significant dimension */

for d := n - 2 downto 0 do if $g_{d+1} = 1$ then exch. content with the neighbor in dim. dendif enddo

The control in the above algorithm is particularly simple, since the following corollary follows from Lemma 3.

Gr	ay code		Exchange dim. 2		Exchange dim. 1		hange dim. 0
data	paddr	ba	data	b2	data	b 1	data
00	0000	0	ÖÖ	0	ŌŌ	Ō	-00
01	0001	0	01	0	01	0	01
02	0011	0	02	0	02	1	03
03	0010	0	03	0	03	1	02
04	0110	0	04	1	07	1	06
05	0111	0	05	1	06	1	07
06	0101	0	06	1	05	0	05
07	0100	0	07	1	04	0	04
08	1100	1	15	1	12	C	12
09	1101	1	14	1	13	0	13
10	1111	1	13	1	14	1	15
11	1110	1	12	1	15	1	14
12	1010	1	11	0	11	1	10
13	1011	1	10	0	10	1	11
14	1001	1	09	0	09	0	09
15	1000	1	08	0	08	0	08

Figure 4: Conversion of a binary-reflected Gray code to binary code

Corollary 4 If the conversion from binary-reflected Gray code to binary code proceeds from the most significant dimension to the least significant dimension, then the current value of b_m assigned to an address is equal to g_m , where m is the controlling dimension.

The algorithm is easy to generalize to an arbitrary starting dimension $m, m \in \mathbb{Z}_{n-1}$ with exchanges in successive dimensions of decreasing order in a cyclic fashion. The first exchange requires the computation of b_m . Figure 5 gives an example. Sequence 2 is the same as in Figure 4. The figure shows the location of i for each step of the algorithm for each sequence. For concurrent exchanges the local data set K is divided into n-1 sets, and set $m, m \in \mathbb{Z}_{n-1}$ is subject to exchange in dimension $(n-2-m-t) \mod (n-2)$ during step t, $t \in \mathbb{Z}_{n-1}$.

```
/* Converting Gray code to binary code starting from dimension m. Dimensions in decreasing order, cyclically*/if g<sub>n-1</sub> ⊕ g<sub>n-2</sub> ⊕ ··· ⊕ g<sub>m+1</sub> = 1 then exch. content with the neighbor in dim. m endif for d := m - 1 downto 0 do if g<sub>d+1</sub> = 1 then exch. content with the neighbor in dim. d endif enddo for d := n - 2 downto m + 1 do if g<sub>d+1</sub> = 1 then exch. content with the neighbor in dim. d endif
```

enddo

Gray	code	I	Seq 2			Seq 1			Seq 0	
Assig	nment	Back	hange	dim.	Back	hange	dim.	Bxc	ange	dim.
Data	paddr	2	i	0	1	ō	2	0	2	1
0	0000	0	0	0	Ö	0	0	0	0	0
1	0001	1	1	1	1	1	1	1	1	1
2	0011	2	2	3	2	3	3	3	3	3
3	0010	3	3	2	3	2	2	2	2	2
4	0110	4	7	6	7	6	6	4	4	•
5	0111	5	6	7	6	7	7	5	5	7
6	0101	6	8	5	Б	5	5	7	7	6
7	0100	7	4	4	4	4	4	6	6	4
8	1100	15	12	12	8	8	12	8	14	12
9	1101	14	13	13	9	9	13	9	18	13
10	1111	13	14	15	10	11	15	11	13	15
11	1110	12	15	14	11	10	14	10	12	14
12	1010	11	11	10	15	14	10	12	10	10
13	1011	10	10	11	14	15	11	13	11	11
14	1001	9	9	9	13	13	9	15	9	9
15	1000	8	8	8	12	12	8	14	8	

Figure 5: Concurrent conversion of a binary-reflected Gray code to binary code.

5 Summary

We have shown that the splitting of a binary-reflected Gray code encoded axis into two binary-reflected Gray coded axes only requires an exchange in the most significant dimension of the lower order axis. The exchanges required for multiple axis splittings can be performed in arbitrary order.

Assume concurrent communication on all ports, K elements per processor, and δ dimensions requiring communication for the reshape operation. If K is a multiple of δ , then the number of element transfers in sequence is independent of δ . An upper bound is K and a lower bound is $\frac{K}{2}$. We present three algorithms: (i) one of communication complexity $\delta \lceil \frac{K}{\delta} \rceil$, (ii) one of complexity $\delta \lceil \frac{K}{2\delta} \rceil + 1$ for reshape operations for which no two dimensions requiring communication are adjacent and $K > 2\delta$, and (iii) and one of complexity $\frac{K}{2} + 2\delta - 1$, if there is one unused processor dimension of higher order for every processor dimension requiring communication. The previously best known algorithm has a complexity of $K + \delta - 1$ [6].

The conversion between binary-reflected Gray code and binary code encodings is a special case of reshaping an array, and can be carried out on an n-cube by n-1 exchanges in dimensions $0,1,\cdots,n-2$ in arbitrary order with a complexity of at most $(n-1)\lceil \frac{K}{n-1} \rceil$ element transfers in sequence.

References

- [1] CM-Fortran Release Notes. Thinking Machines Corp., 1989.
- [2] Jean-Philippe Brunet, Danny C. Sorensen, and S. Lennart Johnsson. A Data Parallel Implementation of the Divide-And-Conquer Algorithm

- for Computing Eigenvalues of Tridiagonal Systems. Technical Report, Thinking Machines Corp., 1989. in preparation.
- [3] J.J. Dongarra and D.C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. SIAM J. Scientific and Statistical Computing, 8(2):s139-s153, 1987.
- [4] Peter M. Flanders. A unified approach to a class of data movements on an array processor. *IEEE Trans. Computers*, 31(9):809-819, September 1982.
- [5] Ching-Tien Ho and S. Lennart Johnsson. Stable Dimension Permutations on Boolean Cubes. Technical Report YALEU/DCS/RR-617, Department of Computer Science, Yale University, October 1988.
- [6] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. J. Parallel Distributed Comput., 4(2):133-172, April 1987. (Tech. Rep. YALEU/DCS/RR-361, Yale Univ., New Haven, CT, January 1985).
- [7] S. Lennart Johnsson. Optimal Communication in Distributed and Shared Memory Models of Computation on Network Architectures, page. Morgan Kaufman, 1989.
- [8] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur. Matrix multiplication on the connection machine. In Supercomputing 89, page, ACM, November 1989. Department of Computer Science, Yale University, Technical Report YALEU/DCS/RR-736.
- [9] S. Lennart Johnsson and Ching-Tien Ho. Reshaping of Arrays on Boolean Cubes. Technical Report, Department of Computer Science, Yale University, 1990. in Preparation.
- [10] S. Lennart Johnsson and Ching-Tien Ho. Shuffle Permutations on Boolean Cubes. Technical Report YALEU/DCS/RR-653, Department of Computer Science, Yale University, October 1988.
- [11] Michael Metcalf and John Reid. Fortran &X Explained. Oxford Scientific Publications, 1987.
- [12] David Nassimi and Sartaj Sahni. Optimal bpc permutations on a cube connected simd computer.

 IEEE Trans. Computers, C-31(4):338-341, April 1982
- [13] David Nassimi and Sartaj Sahni. An optimal routing algorithm for mesh-connected parallel computers. *JACM*, 27(1):6-29, January 1980.
- [14] E M. Reingold, J Nievergelt, and N Deo. Combinatorial Algorithms. Prentice-Hall, Englewood Cliffs. NJ, 1977.

[15] Paul N. Swarstrauber. Multiprocessor FFTs. Parallel Computing, 5:197-210, 1987.

Random Number Generation in the Parallel Environment

Harry F. Sharp, III and Charles H. Still Parallel Supercomputer Initiative University of South Carolina Columbia, S.C. 29208

Abstract

By randomly seeding individual processors in a parallel environment with unique random number generators it is possible to take full advantage of the economies of scale present in the parallel environment to achieve more accurate simulations. While a single random number generator is sufficient for a serial computer, the same is not true for a parallel computer. Multiple copies of the same generator do not improve the quality of the simulation as the period may be insufficient to prevent exhaustion or 'banding' of the variates. Our approach is to provide each processor with its own unique random number generator and use a common seed value. This ensures each simulation is unique as each generator is different due to random assignment by the front-end computer. The linear congruential method was chosen due to widespread familiarity and acceptance of the technique. By using a sequence of random numbers generated on the front-end computer, prime numbers are selected from a predefined array of 2048 primes and assigned to processors. To provide maximum possible period to the generators, all 2048 primes in the array are six digits in size. This gives the researcher the ability to run simulations involving up to a million random numbers with a high degree of certainty that each processor is running a different simulation. By taking advantage of the large periods, the economies of scale available on a parallel machine can then be expolited to run large scale simulations involving millions of numbers which would be prohibitive on a serial machine.

Random Numbers and Simulation

Simulation has achieved a new level of importance in the modern era. Many things of interest to researchers cannot be directly viewed, experimented upon, or actually done due to prohibitive cost or danger. In these cases, simulation becomes the researcher's main tool.

Simulations of real world systems are usually mathematical models. Nuclear plants, national economies, and plane crashes are only a few of the systems which can be modeled by the use of mathematics. However, the formulas themselves are useless without some sort of input to drive them. Random numbers drive the equations and produce the results.

Random numbers are usually the input for these models since the models are based, for the most part, on probabilities. As an example, a splitting atom gives off a neutron, which has a probability k of hitting another atom and forcing it to split. There is also, however, the probability 1-k that the neutron will not hit another atom and the reaction may die.

This method of simulation is known as stochastic simulation but usually referred to as Monte Carlo simulation. Monte Carlo involves sampling from a specific distribution, usually the Uniform (0,1], as these numbers approximate probabilites. The samples are then used to estimate the value of an integral, even in cases where the integral may not be readily apparent. Random numbers must be generated somehow, as computers, where most of the simulations are now done, do not have built-in tables of random numbers. It is for this purpose that the congruential generator methods were developed.

The Linear Congruential Method

For the purpose of this paper, the Linear Congruential Generator method was employed. Motivating our choice of the LCG was ease of programming, debugging, and understanding. The congruential generation method is probably the most commonly used method for generating random numbers today. For a complete explanation of the method see either Kennedy and Gentle [2] or Rubinstein [4]. For those who may not be acquainted with these methods we shall provide a brief overview.

Congruential methods generate pseudo-random numbers by using a recursive formula. The numbers generated are called pseudo-random since they are, per force, deterministic. Given the same formula, the same seed value will always generate the same sequence of "random" numbers. This is not truly unfortunate since it make replicating results possible, which a truly random sequence of numbers would make impossible.

The general form of a congruential generator is

$$x_i \equiv (\alpha x_{i-1} + c) \pmod{m}$$
 for $i = 1, 2, ...$

where

 x_i - the next random in the sequence

 α - the multiplier

c - the increment

m - the modulus

and $0 \le x_i < m$. The value for x at i = 0 is referred to as the seed and is provided by the user. α, c , and m must all be nonnegative.

The generator can have a maximal period, the interval between repeating values, of m if and only if the following conditions are met: [2]

- 1. c is relatively prime to m
- 2. $\alpha \equiv 1 \pmod{p}$ for every prime factor p of m
- 3. $\alpha \equiv 1 \pmod{4}$ if 4 is a factor of m

In reality, finding a quadruple (α, c, p, m) would take too much time to be justifiable. A reasonable approximation for c can be made by

$$c = (1/2 - 1/6 * \sqrt{3}) * m$$

which was proposed by Knuth [3].

The Current Approach

Fox et. al. [1] suggest using the linear congruential generator method, but adapting it to the parallel architecture of a hypercube. The approach they propose is to load each node with the same generator but have nodes "leapfrog" each other. This staggering effect is obtained by having each node step into the sequence of randoms n variates, where n is the processor's number, starting at zero. Formally, if there are p nodes, this means that node 0 gets x_0 as a random value, node 1 gets x_1 , node 2 gets x_2 , ..., node p-1 gets x_{p-1} , node 0 gets x_p , node 1 gets x_{p+1} , and so on

While this accomplishes the task of generating random numbers for each node, and parallelizes the task, it carries some inherent problems.

First, this method uses only one random number generator. While some randomization of the nodes is introduced by the stepping algorithm, the fact that only one sequence of numbers is being used is not overcome. This can have debilitating side effects. If the period is not large enough, the nodes might overlap, producing multiple simulations which are identical. True, this is a pathological example, but even if the period is large enough to prevent overlap, banding will occur in the randoms. By banding, we mean that the points $(x_0, x_1), (x_2, x_3), \ldots$ when plotted produce bands, indicating a high degree of correlation.

It is possible for a simulation to exhaust the period of the random number generator since the period is reduced in size to m/n where n is the number of nodes in a simulation. In such cases, the generator would begin generating the same randoms again, due to the deterministic nature of the algorithm. Since the nodes are staggered, each node would begin to progressively exhaust the period of its generator. Nodes

would then begin to rerun simulations which had already been run by other nodes thereby producing identical results.

Fox's method does provide a way of generating random numbers for parallel simulations, but it is a method open to needless redundancy of effort.

A Second Approach

As an alternative to the method proposed by Fox et. al. we present the following method. Where Fox uses the same random number generator on all nodes, effectively duplicating the same work on all nodes, we provide each node with its own, unique, random number generator. This has the effect of giving each node a different simulation to run, and consequently, a different answer for the simulation. This avoids unecessary duplication of work and reduces the possibility of exhausting the period or "banding" of the random numbers.

Implementation is fairly simple. Large primes are generated and stored on the host computer. By large, we mean primes of at least six digits, the rationale being that the size of the modulus determines the maximum period of the generator and we wish to provide a workable period for large simulations. At load time, the host program selects a prime at random from the stored numbers and passes is to the node being loaded. Selection of the prime is done using a random number generated by the host program. The method of generation for this random is arbitrary and could be accomplished using a builtin random number generator such as rand(). To ensure reproducibility of results, the seed used to initialize the prime selection generator is the same used to seed the individual node random number generators. The prime passed to the nodes is used as the modulus in the LCG, thereby making the random number generator for node i different from the generator for node j.

Results

The method was tested on the NCUBE/Ten here at University of South Carolina. Two different simulations were run: estimation of π and integral estimation. In both cases, the results from the simulations produced values which were correct to two decimal places. Greater precision was not attainable although several attempts were made.

Results from the parallel method were obtained using cubes of size 64, 128, 256, and 512 performing 10000 simulations each. The seed values were taken from a table of primes.

Estimation of π

This test is the same one detailed in Fox. Two random variates are generated from the Uniform (0,1] distribution. The values are squared and then summed. If the result is less than one, a success is recorded. At the completion of the simulation, the total number of successes is multiplied by four and divided by the total number of simulations performed. The result is a rough approximation of π (3.14159).

Table 1 – Estimates of π

Seed	64	128	256	512
7741	3.1379	3.1427	3.1450	3.1462
15017	3.1398	3.1396	3.1396	3.1395
20117	3.1400	3.1400	3.1405	3.1407
31271	3.1457	3.1457	3.1456	3.1456
92857	3.1393	3.1384	3.1380	3.1377

Integral Estimation

This test consisted of two different methods of integral estimation: Monte Carlo and Hit-or-Miss. The target value for both tests in this simulation was 1/3 (.333333).

The Monte Carlo test was done by generating samples of size 200 from the Uniform (0,1] distribution. Variates from the samples were squared and summed and then multiplied by the number of variates in each sample to arrive at the estimation of the integral.

The Hit-or-Miss test used pairs of random variates from the Uniform (0,1] distribution. For each pair, if x_k is less than x_{k-1}^2 then add one to the success count. The total number of successes is then divided by the number of pairs to arrive at the estimate for the integral.

Table 2 - Monte Carlo Estimates

Seed	64	128	256	512
7741	0.3339	0.3336	0.3334	0.3327
15017	0.3316	0.3312	0.3310	0.3302
20117	0.3338	0.3338	0.3338	0.3331
31271	0.3323	0.3320	0.3320	0.3313
92857	0.3341	0.3341	0.3341	0.3341

Table 3 - Hit-or-Miss Estimates

Seed	64	128	256	512
7741	0.3332	0.3324	0.3320	0.3312
15017	0.3367	0.3378	0.3383	0.3379
20117	0.3336	0.3335	0.3334	0.3328
31271	0.3375	0.3382	0.3386	0.3381
92857	0.3324	0.3326	0.3327	0.3327

Execution Time

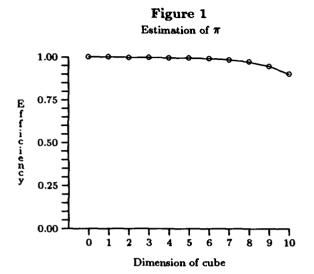
Execution time was averaged for the first test (estimation of π). The timing results are in the following table. Note that the times do not include startup and that there is no node-to-node communication which leads to very high efficiencies. All efficiency estimates were calculated using

efficiency =
$$\frac{\text{time}_1}{p * \text{time}_p}$$
.

Table 4 - Execution Times

Dimension	Time (sec)	Efficiency
0	281.8257	1.0000
1	140.9265	0.9999
2	70.6814	0.9968
3	35.3558	0.9964
4	17.7055	0.9948
5	8.8612	0.9939
6	4.4454	0.9906
7	2.2376	0.9840
8	1.1336	0.9712
9	0.5816	0.9464
10	0.3055	0.9009

These results are from simulations of size 512,000. Thus in the dimension d case, each node generated $\frac{512000}{2^d}$ random numbers. The figure below shows the efficiencies from the above table.



Introduction of Shuffling

While the LCG method is sufficient, it is not always optimal. In many cases, a marginal, or even bad, random number generator can be improved with the addition of a shuffling algorithm such as that developed by Bays and Durham (see [2]). In this case, the LCG method generated variates that were sufficiently random to produce acceptable, if not outstanding, results. After the inclusion of the Bays-Durham shuffling algorithm to further randomize the variates, the results improved. Almost uniformly, the precision of the results increased.

Table 5 – Estimates of π with Shuffling

Seed	64	128	256	512
7741	3.1406	3.1439	3.1454	3.1463
15017	3.1380	3.1387	3.1390	3.1392
20117	3.1412	3.1408	3.1412	3.1414
31271	3.1434	3.1433	3.1433	3.1432
92857	3.1387	3.1381	3.1379	3.1377

Conclusions

What we have shown with this simple example is that, while no dramatic improvement is evident in accuracy, the parallel method is as good as the serial method. Since there is no loss of precision due to the parallelization of the simulation, the researcher could take advantage of the economies of scale inherent in the parallel processor to run truly large simulations which would be prohibitive on a serial machine. This technique also provides a method by which the accuracy of simulations can be tested by running multiple copies in the time it would take to run a single copy.

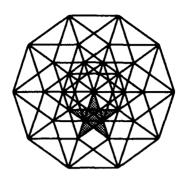
It should be possible to overcome the precision threshold and thereby achieve more accurate results from the parallel simulations. Even with the threshold, however, the results indicate that reliably independent simulations can be run on parallel machines using independent random number generators.

Acknowledgements

We would like to thank Chuck Baldwin for his assistance in debugging the code and helping with the communications problems on the Cube.

References

- [1] Fox, Geffery C., et. al. Solving Problems on Concurrent Processors: Volume 1 - General Techniques and Regular Problems. Prentice Hall, Englewood Cliffs, NJ. 1988.
- [2] Kennedy, Jr., William J. and James E. Gentle. Statistical Computing. Marcel Dekker, Inc., New York, NY. 1980.
- [3] Knuth, Donald E. The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison-Wesley, Reading, Massachusetts, 1969.
- [4] Rubinstein, Reuven Y. Simulation and the Monte Carlo Method. John Wiley & Sons, New York. 1981.



The Fifth Distributed Memory Computing Conference

15: Monte Carlo Physics

Cluster Algorithms for Spin Models on MIMD Parallel Computers

Paul D. Coddington and Clive F. Baillie

Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, CA 91125, USA

Abstract

Parallel computers are ideally suited to the Monte Carlo simulation of spin models using the standard Metropolis algorithm, since it is regular and local. However local algorithms have the major drawback that near a phase transition the number of sweeps needed to generate a statistically independent configuration increases as the square of the lattice size. New algorithms have recently been developed which dramatically reduce this 'critical slowing down' by updating clusters of spins at a time. The highly irregular and non local nature of these algorithms means that they are much more difficult to parallelize efficiently. Here we introduce the new cluster algorithms, explain some sequential algorithms for identifying and labelling connected clusters of spins, and then outline some parallel algorithms which have been implemented on MIMD machines.

1. Introduction

Computer simulations are extremely useful in the study of spin models in condensed matter physics. In these models the spins are usually set up on the sites of a d-dimensional hypercubic lattice of length L. The L^d spins form some configuration. The goal of computer simulations is to generate configurations of spins typical of statistical equilibrium and measure physical observables on this ensemble of configurations. This is traditionally performed by Monte Carlo methods such as the Metropolis algorithm [1], which produce configurations with a probability given by the Boltzmann distribution $e^{-\beta S(\phi)}$, where $S(\phi)$ is the action, or energy, of the system in configuration ϕ , and β is the inverse temperature. One of the main problems with these methods in practice is that successive configurations

are not statistically independent, but rather are correlated, with some autocorrelation time τ between effectively independent configurations.

A key feature about traditional (Metropolislike) Monte Carlo algorithms is that the updates are local, since the procedure to update a given spin depends only on the values of spins which affect its contribution to the action, and most spin models have local (usually nearest neighbor) interactions. Thus in a single step of the algorithm, information about the state of a spin is transmitted only to its near neighbors. In order for the system to reach a new effectively independent configuration, this information must travel a distance of order the (static or spatial) correlation length ξ . As the information executes a random walk around the lattice, one would suppose that the autocorrelation time $\tau \sim \xi^2$. However, in general $\tau \sim \xi^z$, where z is called the dynamical critical exponent. Almost all numerical simulations of spin models have measured $z \approx 2$ for local update algorithms.

For a spin model with a phase transition, as the inverse temperature β approaches its critical value, ξ diverges to infinity, so that the computational efficiency rapidly goes to zero! This behavior is called critical slowing down (CSD), and until very recently it has plagued Monte Carlo simulations of statistical mechanical systems, in particular spin models, at or near their phase transitions. Recently, however, several new 'cluster algorithms' have been introduced which decrease z dramatically by performing non-local spin updates, thus reducing (or even eliminating) CSD and facilitating much more efficient computer simulations.

Parallel computers have been very successfully applied to the Monte Carlo simulation of spin models using the traditional algorithms such as that of Metropolis. These algorithms are easily and efficiently parallelised using domain decomposition of

the lattice, since they are very regular (and hence perfectly load balanced), and only require a small amount of local communication between processors. The new cluster algorithms, on the other hand, are highly irregular and non-local, and are therefore much more difficult to parallelize efficiently. Here we introduce the cluster update algorithms, explain some sequential algorithms for identifying and labeling connected clusters of spins, and then outline some parallel algorithms which have been implemented on MIMD machines.

2. Cluster algorithms

The aim of the cluster update algorithms is to find a suitable collection of spins which can be flipped with relatively little cost in energy. We could obtain non-local updating very simply by using the standard Metropolis Monte Carlo algorithm to flip randomly selected bunches of spins, but then the acceptance would be tiny. We need a method which picks sensible bunches or clusters of spins to be updated. The first such algorithm was proposed by Swendsen and Wang [2], and was based on an equivalence between a Potts spin model [3] and percolation models [4], for which cluster properties play a fundamental role.

The Potts model is a very simple spin model of a ferromagnet, in which the spins can take q different values. The case q=2 is just the well-known Ising model. In the Swendsen and Wang algorithm, clusters of spins are created by introducing bonds between neighboring spins with probability $1-e^{-\beta}$ if the two spins are the same, and zero if they are not. All such clusters are created, and then updated by choosing a random new spin value for each cluster and assigning it to all the spins in that cluster.

A slightly different cluster algorithm has been proposed by Wolff [5]. In this algorithm, a spin is chosen at random and a single cluster constructed around it, using the same bond probabilities as for the Swendsen-Wang algorithm. All the spins in this cluster are then flipped (i.e. changed to a random new spin different from the old one). Although Wolff's method is probably the best sequential cluster algorithm, the Swendsen and Wang algorithm seems to be better suited for parallelization, since it involves the entire lattice rather than just a single cluster. We have therefore concentrated our attention on the method of Swendsen and Wang, where all the clusters must be identified and labeled.

3. Cluster identification

Cluster algorithms have in common the problem of identifying and labeling the connected clusters of spins. This is very similar to an important problem in image processing, that of identifying and labeling the connected components in a binary or multi-colored image composed of an array of pixels. The only real difference is that in the spin model case, neighboring sites of the same spin have a certain probability of being in the same cluster, while for neighboring pixels of the same color that probability is one. Unfortunately this is a large enough difference so that some algorithms which work in image analysis will not work (or require substantial changes) for spin models.

First we mention some sequential algorithms for labeling clusters of connected sites. Perhaps the most obvious method for identifying a single cluster is the so-called 'ants in the labyrinth' algorithm. The reason for its name is that we can visualize the algorithm as follows [6]. An ant is put somewhere on the lattice, and notes which of the neighboring sites are connected to the site it is on. At the next time-step this ant places children on each of these connected sites which are not already occupied. The children then proceed to reproduce likewise until the entire cluster is populated. In order to label all the clusters, we start by giving every site a negative label, set the initial cluster label to be zero, and then loop through all the sites in turn. If a site's label is negative then the site has not already been assigned to a cluster, so we place an ant on this site, give it the current cluster label, and let it reproduce, passing the label on to all its offspring. When this cluster is identified we increment the cluster label and carry on, repeating the ant-colony birth, growth and death cycle until all the clusters have been identified.

An alternative method which is commonly used (especially for cluster identification in percolation models) is that of Hoshen and Kopelman [7]. We have found that 'ants' gives slightly better performance than this algorithm, and so we will not discuss it further.

Identifying and labeling clusters of connected sites in a lattice is a special case of the more general problem known variously as the set union, union-find, or equivalence problem, i.e. given a list of equivalences between elements, sort the elements into equivalence classes. In the context of cluster algorithms, the list of equivalences is just a list of the sites which are connected together, and the equivalence classes are just the clusters. There are a multitude of algorithms for this problem [8]. We have used an elegant and easy to code method due to Galler and Fisher [9], which goes as follows. Let F(j) be the class or 'family' label of element j. We

start off with each element in its own family, so that F(j) = j. The array F(j) can be interpreted as a tree structure, where F(j) denotes the parent of j. If we arrange for each family to be its own tree, disjoint from all other 'family trees', then we can label each family by its most senior great-great-...grandparent. Therefore we process each equivalence of two sites j and k by

- (1) tracking j up to its highest ancestor,
- (2) tracking k up to its highest ancestor,
- (3) giving j to k as a new parent

After processing all the equivalence relations, we go through all the elements j and reset their F(j)'s to their highest possible ancestors, which then label the equivalence classes, so that F(j) is the cluster label of site j.

4. Parallel algorithms

As with the percolation models upon which the cluster algorithms are based, the phase transition in a spin model occurs when the clusters of bonded spins become large enough to span the entire lattice [4]. Thus near criticality (which in most cases is where we want to perform the simulation) clusters come in all sizes, from order N (where N is the number of sites in the lattice) right down to a single site. The highly irregular and non-local nature of the clusters means that cluster update algorithms do not vectorize, and hence give poor performance on vector machines. On this problem a CRAY X-MP is only about ten times faster than a SUN4 workstation. SIMD machines are similarly unsuited to this problem, whereas for the Metropolis type algorithms they are perhaps the best machines available. It therefore appears that the optimum performance for this type of algorithm will come from MIMD parallel computers.

Using the trivial parallelization technique of running independent Monte Carlo simulations on different processors, it is possible to do better than a CRAY on a typical MIMD parallel computer with only about 20 nodes. This method works well until the lattice size gets too big to fit into the memory of each node, and in fact we have used this method to calculate the dynamical critical exponents of various cluster algorithms [10]. However in the case of the Potts model, for example, only lattices of size less than about 300^2 or 50^3 will fit into 1 Mbyte, and most other spin models are more complicated and more memory intensive. We therefore need a parallel algorithm where a large lattice can be distributed over many processors.

A parallel cluster algorithm involves distributing the lattice onto an array of processors using the usual domain decomposition. Clearly a sequential algorithm can be used to label the clusters on each processor, but we need a procedure for converting these labels to their correct global values. We need to be able to tell many processors, which may be any distance apart, that some of their clusters are actually the same. Thus we need to be able to agree on which of the many different local labels for a given cluster should be assigned to be the global cluster label, and to pass this label to all the processors containing a part of that cluster. We will discuss two methods for tackling this problem, 'self-labeling' and 'global equivalencing', and briefly mention some other algorithms for labeling clusters in parallel.

4.1. Self-labeling

We shall refer to this algorithm as 'self-labeling', since each site figures out which cluster it is in by itself from local information. We begin by assigning each site i a unique cluster label S_i . In practice this is simply chosen as the position of that site in the lattice. At each step of the algorithm, in parallel, every site looks in turn at each of its neighbors in the positive directions. If it is bonded to a neighboring site n which has a different cluster label S_n , then both S_i and S_n are set to the minimum of the two. This is continued until nothing changes, by which time all the clusters will have been labeled with the minimum initial label of all the sites in the cluster. Note that to check termination of the algorithm involves each processor sending a termination flag (finished or not finished) to every other processor after each step, which can become very costly for a large processor array.

We can improve this method by using a faster sequential algorithm, such as 'ants in the labyrinth', to label the clusters in the sublattice on each processor, and then just use self-labeling on the sites at the edges of each processor to eventually arrive at the global cluster labels. The number of steps required to do the self-labeling will depend on the largest cluster, which at the phase transition will generally span the entire lattice. The number of self-labeling steps will therefore be of the order of the maximum distance between processors, which for a square array of P processors is just $2\sqrt{P}$. Hence the amount of communication (and calculation) involved in doing the self-labeling, which is proportional to the number of iterations times the perimeter of the sublattice, goes like L for an LxL lattice, whereas the time taken on each processor to do the local cluster

labeling goes like the area of the sublattice, which is L^2/P . Therefore as long as L is substantially greater than the number of processors we can expect to obtain a reasonable speedup.

The speedups obtained on the Symult 2010 for a variety of lattice sizes are shown in Fig. 1. The dashed line indicates perfect speedup (i.e. 100% efficiency). The lattice sizes for which we actually need large numbers of processors are of the order of 512² or greater, and we can see that running on 64 nodes (or running multiple simulations of 64 nodes each) gives us quite acceptable efficiencies of about 70% for 5122 and 80% for 10242. In Table 1 we show a comparison of times for one update of a 5122 lattice using self-labeling on various MIMD parallel computers, and compare this with results for the fastest algorithm on a SUN workstation and a CRAY X-MP. The time on the CRAY is taken from Wolff [11]. Note that using all 512 nodes of Caltech's NCUBE, by running multiple 64 node simulations, gives a performance approximately five times that of the CRAY, while all 192 nodes of the Symult S2010 is equivalent to about six CRAYs for this problem.

Machine	Nodes	Time (sec)
SUN-4	1	16.0
CRAY X-MP	1	1.7
NCUBE/1	64	2.8
Symult	64	0.82
Meiko	32	1.2

Table 1. Times for one update of a 512² lattice using the Swendsen and Wang cluster algorithm. Self-labeling is used on the parallel machines.

4.2. Global equivalencing

In this method we again use the fastest sequential algorithm to identify the clusters in the sublattice on every node. Each node then checks to see which of the edge sites of its sublattice are connected to edge sites on the neighboring nodes in the positive directions, and are therefore part of the same cluster and should be given the same cluster label. These lists of 'equivalences' are all passed to one of the nodes, which uses the equivalence class algorithm of Fisher and Galler [9] to match up the connected subclusters, and then broadcasts the global cluster labels to all the other nodes.

The problem here is that the equivalencing is purely sequential, and is thus a potentially disastrous bottleneck for large numbers of processors. The amount of work involved goes like the number of processors P times the perimeter of the sublattice on each node, so that the efficiency should be less

than for self-labeling, although we might still expect reasonable speedups if the number of nodes is not extremely large. The speedups obtained for this algorithm on the Symult 2010 for a variety of lattice sizes are shown in Fig. 2. Global equivalencing gives about the same speedups as self-labeling for small numbers of processors, but as expected self-labeling does much better as the number of nodes increases.

To get around this sequential bottleneck we need to adopt a hierarchical divide-and-conquer approach, where the equivalence classes are built up in stages. In this approach the processor array is divided up into smaller subarrays of, for example, 2x2 processors. In each subarray, the processors look at the edges of their neighbors for clusters which are connected across processor boundaries. These equivalences are all passed to one of the nodes of the sub-array, which places the cluster labels in equivalence classes as before. The results of these partial matchings are similarly combined across the edges of each 4x4 subarray, and this process is continued until finally all the partial results are merged together on a single processor to give the global cluster values, which are then passed back through the hierarchy of levels. This type of algorithm has been implemented on a hypercube for the image processing component labeling problem by Embrechts et al. [12]. We are currently implementing the hierarchical global equivalencing algorithm for the spin model case, which should do better than selflabeling for large numbers of processors, since the number of steps required goes like the logarithm (rather than the square root) of the number of processors.

4.3 Other algorithms

Currently the only other parallel cluster algorithm implemented for spin models is a parallel extension of the Hoshen and Kopelmann algorithm [7] due to Burkitt and Heermann [13], but it is much more complicated, and less efficient, than the selflabeling algorithm. There have been many different parallel algorithms proposed for the connected component labeling problem in image analysis. Some of these algorithms are aimed at shared memory [14] or SIMD [15] [16] architectures, but could probably be implemented on distributed memory MIMD machines also. Others are based on MIMD machines such as hypercubes [17]. These algorithms also need investigation to see if they might be applied to the problem of producing a more efficient parallel cluster algorithm for spin models on large numbers of processors.

References

- [1] N. Metropolis et al., J. Chem. Phys. 21, 1087 (1953).
- [2] R.H. Swendsen and J.-S. Wang, Phys. Rev. Lett. 58, 86 (1987).
- [3] R.B. Potts, Proc. Camb. Phil. Soc. 48, 106 (1952); F.Y. Wu, Rev. Mod. Phys. 54, 235 (1982).
- [4] D. Stauffer, Phys. Rep. 54, 1 (1978); J.W. Essam, Rep. Prog. Phys. 43, 833 (1980).
- [5] U. Wolff, Phys. Rev. Lett. 62, 361 (1989).
- [6] R. Dewar and C. K. Harris, J. Phys. A 20, 985 (1987).
- [7] J. Hoshen and R. Kopelman, Phys. Rev. B14, 3438 (1976).
- [8] R.E. Tarjan and J. van Leeuwen, J. ACM 31, 245 (1984).
- [9] D.E. Knuth, Fundamental Algorithms, vol. 1 of The Art of Computer Programming (Addison-Wesley, Reading, 1968); W.H. Press et al., Numerical Recipes in C; The Art of Scientific Programming, (Cambridge University Press, Cambridge, 1988).
- [10] C.F. Baillie and P.D. Coddington, "A Comparison of Cluster Algorithms for Potts Models", Caltech Concurrent Computation Report C³P-835, October 1989.
- [11] U. Wolff, Phys. Lett. B228, 379 (1989).
- [12] H. Embrechts and D. Roose, "Efficiency and Load Balancing Issues for a Parallel Component Labeling Algorithm", Proc. Fourth Conference on Hypercubes, Concurrent Computers and Applications, Monterey, 1989.
- [13] A.N. Burkitt and D.W. Heermann, Comp. Phys. Comm. 54, 210 (1989).
- [14] R. Hummel, "Connected component labeling in image processing with MIMD architectures", in *Intermediate-Level Image Processing*, (Academic Press, New York, 1986).
- [15] W. Lim, A. Agrawal, L. Nekludova, "A Fast Parallel Algorithm for Labeling Connected Components in Image Arrays" Thinking Machines Corporation Technical Report NA86-2.
- [16] D. Nassimi and S. Sahni, SIAM J. Comput. 9, 744 (1980).
- [17] R. Cypher, J.L.C. Sanz and L. Snyder, J. Algorithms 10, 140 (1989).

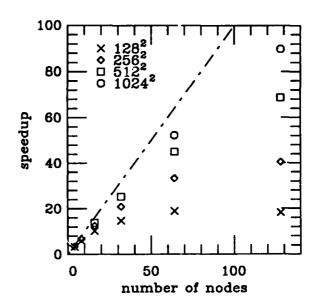


Fig. 1. Speedups for self-labeling on the Symult S2010.

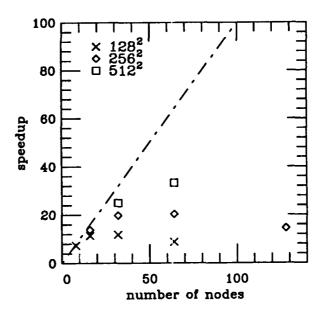


Fig. 2. Speedups for global equivalencing on the Symult S2010.

II.-Q. Ding and M. S. Makivic

Concurrent Computation Program and Physics Department California Institute of Technology, Pasadena, CA 91125

Abstract

A large scale Quantum Monte Carlo simulation is performed on the Mark IIIfp Hypercube supercomputer to systematically study the quantum spin dynamics of the recently discovered high- T_c superconducting mother material. The algorithm is very efficiently implemented on the Hypercube. The 3dimensional lattice is partitioned into a ring of processor nodes. Parallelism is also achieved by running several independent simulations on several processor rings simultaneously. The local updates are easily handled by the CROS communication system. Global updates are efficiently implemented by a "gather - scatter" routines written in CROS calls. Spins are packed into 32-bit words along the time direction. Local updates are vectorized along time direction. We also report a systematic performance analysis. The efficiency of the implementation is over 90%.

1. Introduction

The power of parallel computers is doubling each six month in recent years. Significant computations[1] in scientific and engineering researches are performed on these parallel computers. Many of the applications[2] easily achieved performance better (in some cases, much better) than the conventional supercomputers and obtained new, important results. In this paper, we report such an application on the MarkHIfp hypercube[3,4].

The discovery of high- T_c superconductors[5] has led to enormous amount of research on the two-

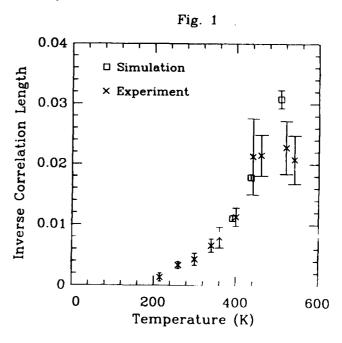
dimensional magnetism. Magnetic properties are believed to play significant role in the new mechanism for the high- T_c superconductivity. These magnetism is essentially modeled by the quantum antiferromagnetic Heisenberg model:

$$H = J \sum_{\langle ij \rangle} (S_i^x S_j^x + S_i^y S_j^y + S_i^z S_j^z)$$

where $S^a = (1/2)\sigma^a$ are quantum spin operators:

$$\sigma^x = \begin{vmatrix} 0 & 1 \\ 1 & 0 \end{vmatrix} \quad \sigma^x = \begin{vmatrix} 0 & -i \\ i & 0 \end{vmatrix} \quad \sigma^x = \begin{vmatrix} 1 & 0 \\ 0 & -1 \end{vmatrix}.$$

This model is also an limiting case of a much more general theory. We have performed a series of large scale quantum Monte Carlo simulations on these



models and obtained significant results. We only mention here that our data agrees with the neutron

scattering experiments very well and this calculation is the first to give an accurate first principles determination of the exchange coupling $J=1450\pm30K$ and spin stiffness constant $\rho=0.199(2)$. Fig.1 shows the comparison between our data and the experiments. More details of the physics results is reported in [6]. On a related quantum XY model, we found[7] a Kosterlitz-Thouless phase transition thus completes more than 20 years investigation into an important problem in statistical physics.

In this paper, we will concentrate on the computational aspects of the simulation. We first outline the algorithm. The multi-coding technique is explained next. Parallel implementation is then discussed in detail. We give a systematic performance analysis.

2. Converting Quantum Problem to Classical Problem

The Monte Carlo algorithm we used is a fairly standard one in statistical physics, although the details are quite complex due to the quantum conservation laws. Following the Suzuki-Crotter approach we first convert the quantum problem into a classical one. The partition function for the Heisenberg model Eq.1 on the 2- dimensional lattice can be rewritten as:

$$Z={\rm Tr}e^{-\beta H}={\rm Tr}(e^{-\beta H/m})^m=$$

$$\lim_{m\to\infty}{\rm Tr}(e^{-\beta H_1/m}e^{-\beta H_2/m}e^{-\beta H_3/m}e^{-\beta H_4/m})^m$$
 where $\beta=1/kT$ and

$$H = H_1 + H_2 + H_3 + H_4$$

is a breakup so that each H_i contains only terms commuting among themselves. The integer, m_i is set to be a large but finite number, in practice. After inserting complete sets of states (eigen-states of S_i^z), the partition function breaks down into products of Boltzmann factors associated with interacting 4-spin squares:

$$Z = \lim_{m \rightarrow \infty} \sum_{\{C\}} \langle 1, 1|e^{-\beta H_1/m}|1, 2\rangle \langle 1, 2|e^{-\beta H_2/m}|1, 3\rangle$$

$$\langle 1, 3|e^{-\beta H_3/m}|1, 4\rangle\langle 1, 4|e^{-\beta H_4/m}|2, 1\rangle \cdots$$

$$(m, 1|e^{-\beta H_1/m}|m, 2)(m, 2|e^{-\beta H_2/m}|m, 3)$$

$$\langle m, 3|e^{-\beta H_3/m}|m, 4\rangle\langle m, 4|e^{-\beta H_4/m}|1, 1\rangle$$

This becomes a general classical Ising spin system in 3 dimensions. The Boltzmann weight, associated with a 4-spin square configuration is given by the following transfer matrix:

$$W = \langle S_{i,t}^{z} S_{j,t}^{z} | e^{-(\beta J/m)S_{i} \cdot S_{j}} | S_{i,t+1}^{z} S_{j,t+1}^{z} \rangle$$

more explicitly,

$$W = \begin{bmatrix} e^{K} & 0 & 0 & 0 \\ 0 & e^{-K}ch(2K) & e^{-K}sh(2K) & 0 \\ 0 & e^{-K}sh(2K) & e^{-K}ch(2K) & 0 \\ 0 & 0 & 0 & e^{K} \end{bmatrix}$$

where $K = \beta/4m$. The zero elements in transfer matrix are the consequence of the quantum conservation law. To avoid generating trial configuration with these zero transfer probability thus wasting the CPU time because these trials will never be accepted, one should have the conservation law built into the flipping scheme. We have designed a set of four elementary updates[6] that can generate all possible spin configurations.

3. The Computational Algorithm

This classical spin system in 3 dimensions is simulated using the Metropolis Monte Carlo algorithm. Starting with a given initial configuration, we locate a closed loop C of L spins. After check that they satisfy the conservation law if we flip them all together, we compute the probability that the present configuration remains unchanged:

$$P_i = \prod_{k=1}^{k=L} W_{C_k,C_k}^{(k)}$$

where $W_{C_k,C_k}^{(k)}$ are the diagonal elements of the transfer matrix, and the probability that the configuration are flipped:

$$P_f = \prod_{k=1}^{k=L} W_{C_k, C_{k+1}}^{(k)}$$

where $W_{C_k,C_{k+1}}^{(k)}$ are the off-diagonal elements of the transfer matrix along the loop C. The Metropolis

procedure is to accept the flip according to the probability

$$P = P_f/P_i$$
.

If the flip is not accepted, we keep the initial configuration and go on to the next loop of spins.

The classical system is defined on a 3- dimensional lattice. On each grid point, a spin can only have two states, up or down, which is represented by 0 or 1. An elementary square of 4 spins is called an interacting square if they are connected through the Boltzmann factor. They are denoted by the shaded square in Fig.2. Note that not all squares are interacting.

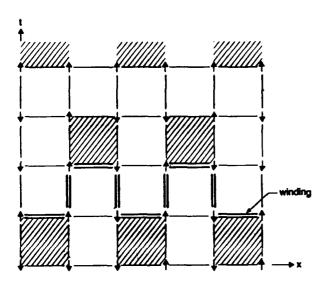
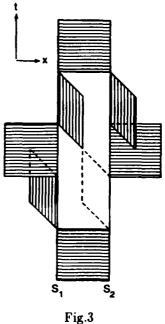


Fig.2

Two types of local moves may locally change the spin configurations. The time-loop local update is shown in Fig.3. (Note that the spins at the lattice sites in Figs. 3-5 are omitted for simplicity. Their presence are same as in Fig.2.) All 8 spins in the loop are either all flipped, or remains unchanged depending on whether the probability test is success or failure. Similarly in the space-loop local update, shown in Fig.4, all 4 spins in the loop are either all flipped or not.

A global move in the time direction, we called it time-line, flips all the spins along this time-line. This update changes the magnetization. Another global move in spatial directions, winding-line, as shown in Fig. 2 by the double line, changes the winding numbers.



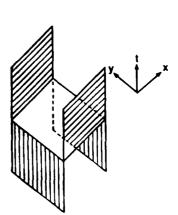


Fig.4

Periodic boundary conditions are imposed in all directions to preserve the translation invariance and to satisfy the trace requirement. In each Monte Carlo sweep through the lattice, we apply the all four moves to all possible configuration. After enough sweeps, the system reaches the equilibrium state, and we then take measurements.

4. The multispin Coding

We implemented a simple and efficient multispin coding method, which facilitates vectorization, saves index calculation and memory space. This is possible because each spin only has two states, up (1) or down (0), which is represented by a single bit in a 32-bit integer. Spins along t- direction is packed into a 32-bit words, so that the boundary communication along x or y direction can be handled more easily.

All the necessary checks and updates can be handled by the bitwise logical operations OR, AND, NOT, XOR. Note that this is a natural vectorization since AND operations for the 32 spins are carried out in the single AND operation by the CPU. The index calculations to address these individual spins are also minimized, because one only computes the index once for the 32 spins. The same principles are applied for both local and global moves, but it is easier to illustrate them for local moves, as shown in Fig.5.

A pair of adjacent words contains eight "time" loops, as indicated by the dotted line in Fig.5. Because every two adjacent "time" loops share an interacting square, we update all four odd "time" loops simultaneously in a vectorized fashion. The other four even "time" loops are updated next. Many of the useful quantities obtained in u; dating the four odd ones will also be used for the four even ones. We now briefly illustrate the scheme. We want to update the odd "time" loops 1, 3, 5 and 7 of the spin words S1 and S2 in Fig.5. We first compute F = S1 [XOR] S2, and then W = F[AND] MASK1, where MASK1 has "1"s located at the proper position of the "time" loop: MASK1=(0...01111). The flip of "time" loop 1 is allowed if W [AND] MASK1 = MASK1 and (S1 [AND] MASK1) + (S2 [AND] MASK1) = 16 (which means that all four spins in S1 must be down and the four spins in S2 must be up, or vice versa). S1 is also XOR-ed with N1, N6 and N5 to obtain E1, E6 and E5, the information needed to compute the energy due to the three interacting loops on S1 side (see Fig.3). Similarly, S2 is XOR-ed with N2, N3 and

N4. Finally, S1 is XOR-ed with (S1 [right-shift] 1) to obtain C which contains the information about the upper and lower interacting loops (which are shared with adjacent "time" loops). After masking N1-N6 and C with appropriate masks, we SHIFT, OR them together, to obtain X1 and X2 which contain the information about the eight interacting loops shared by S1 and S2. Notice that N1-N6 are used only once for all of the eight "time" loops.

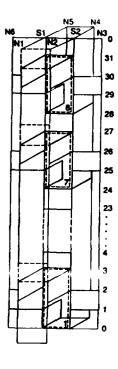


Fig.5

To retrieve the information that pertains to the "time" loop 1, we calculate I1 = X1 [AND] MASK1 and I2 = X2 [AND] MASK1. (I1,I2) is a pair of small integers in one-to-one correspondence with the spin configuration: it uniquely determines the transition probability. Thus (I1,I2) is used as an index to fetch the transition probability stored in a small lookup table calculated at the beginning. The floating point operations in the update are the Metropolis accept/reject test. Upon acceptance, the proper four spins in S1 are flipped by S1 = S1 [XOR] MASK1, and similarly for S2. The update of the "time" loop 3 proceeds in the same manner as for loop 1, after left-shifting MASK1 for 8 bits, and similarly for loops

5 and 7. Once "time" loops 1.3,5,7 are completed, we need to recalculate C only, and the entire process is repeated for even loops. Notice that only floating point operation in these updates is a random number generation[9] and comparison.

Four adjacent words contain eight "space" lo-ops. They can be updated without alternating even and odd ones, since they are decoupled.

The global move in time direction is very easy to implement with this type of spin packing. One has to check whether bits are all either 0's or 1's, then to XOR the word to be flipped with four neighboring words to get the transition probability. The same principles are used to implement the global flip in spatial directions, but the actual procedure is much more complicated. It is desirable to have the simplest possible spin interaction in order to minimize the complexity of the various tests needed to determine the transition probability. For this reason, we believe that our "bond-type" decomposition is preferable due to the simplicity of spin interactions, although the spin packing could be done with any other decomposition, such as "cell-type" breakup, which leads to more complicated 8-spin interactions.

5. Parallel Implementation

Using the multicoding technique, the spins do not occupy large memory spaces, the most simply way is to run an independent simulation on each node and average them to get the statistical results. However, this naive implementation does not work well for this problem. Our lattices is fairly large (128x128x192) in the sense that a typical thermal relaxation take about 10,000 sweeps (details depends on the temperature and the correlation length of the system). Both thermal and quantum fluctuation make the averaging process very long, in units of 100 hours, to obtain a sufficient complete sample. We need to do physical space para lelization to cut this CPU time down to more reasonable time.

We partition the 3- dimensional lattice Nx*Ny *Nt into a ring of M processor nodes so that each

node contain a subspace (Nx/M)*Ny*Nt, shown in Fig.6. The local updates are easily parallelized since the connection is at most next-nearest neighbor (for the time-loop update). The needed spin-word arrays from its neighbor are copied into the the local storage by the shift routine in CROS communication system[3,4] before doing the update. One of the global update, the time-line, can also be done in the same fashion. The communication is very efficient in the sence that a single communication shift Ny*Nt spins instead of Nt spins in the case the lattice is partitioned into 2- dimensional grid. The overhead associated with the communication routine, which is quite non-negligible, is reduced greatly because it only occurs once, instead of Ny times. This is one of the reasons that a 1-d grid decomposition is bether than 2-d decomposition for this class of problems.

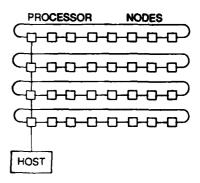


Fig.6

The winding-line global update along x- direction is difficult to do in this fashion, because it involves spins on all the M nodes. In addition, We need to compute the correlation functions which have the same difficulty. However, since these operations are not used very often (every 10 sweep, one may call the winding-line update and the correlation function measurements), we devised a fairly elegant way to parallelize these global operations.

We have wrote a set of gather-scatter routines based on the cread and cwrite in CROS. In gather,

the subspaces on each node are gathered into complete spaces on a each node, preserving the original geometric connection. As shown in Fig.7, each node has a copy, but the x-coordinate is rotated accordingly so the node's own subspace is in the starting position. Parallelism is achieved now since the global operations are done on each node just as in the sequential computer, with each node only do the part it originally covers. In scatter, the updated (changed) lattice configuration on a particular node (number zero) is scattered (distributed) back to all the node in the ring, exactly according the original partition. Note that this scheme differs from the earlier decomposition scheme[8] for the gravitation problem, where memory size constraints is the main concern.

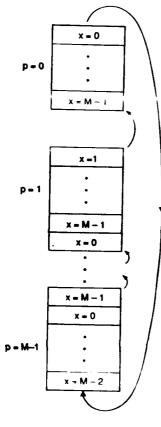


Fig.7

All of our simulations were done on a 32-node hypercube. For higher temperatures, no need to use lattice of sizes of 96x96, or not even 64x64. So the 32 nodes was divided into several independent

rings, each ring holds an independent simulation, as shown in Fig.6. Typically, for 32x32 lattice, we run 8 simulations, each using 4 node-ring. For 96x96 lattice, we run 2 independent runs, each uses 16 node rings. This simple parallelism make the simulation very flexible and efficient. In the simulation, we used a parallel version of the Fibonacci additive random numbers generator[9] which has a period larger that 2^{127} .

6. Performance Analysis

We have made a systematic performance analysis, by running the code on different sizes and different number of nodes. The timing results for a realistic situation (20 sweeps of update, 1 measurement) are given in Table 1. The speedup, t_1/t_M ,

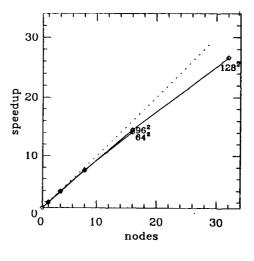


Fig.8

where t_1 (t_M) is the time for the same size spins system to run same number operations on 1 (M) nodes, are listed in Table 1. It is also plotted in Fig.8. One can see that speedup is quite close to the ideal case denoted by the dashed line in Fig.8 For the 128×128 quantum spin system, the 32-node hypercube speedup the computation by a factor of 26.6, a very good result. However, running the same spin system on 16-node is more efficient, because we can run two independent systems on the 32-node hypercube with a total speedup $2 \times 14.5 = 29$ (each speeds up

a factor 14.5). This is better described by efficiency, defined as speedup/nodes, which is tabulated in Table 1. Clearly, the efficiency of the implementation is very high, generally over 90%.

The Exact direct comparison with other supercomputers are not available at present. However, an very similar multispin spin code[10] in calculating the elementary excitation energy spectrum of this same Heisenberg model is running on both MarkIIIfp and on Cray XMP. The Cray speed is approximately equivalent to 2-node MarkIIIfp. This indicates that our 32-node MarkIIIfp performs better than Cray XMP about a factor of (32/2)*90% = 14! We note that our code is written in "C" and the vectorization is limited to the 32-bit inside the words. Rewriting the code in Fortran (Fortran compilers on Cray are more efficient) and fully vectorize the code, one may gain a factor about 5 on Cray. But even after such an big programming efforts on the Cray, MarkIIIfp will probably run faster than Cray XMP by a factor of 3. Clearly this quantum Monte carlo code is a good example in that parallel computers easily (i.e., at same programming level) outperform the conventional supercomputers.

7. Conclusion and Acknowlegements

An implementation of the quantum Monte Carlo code for the spin system on the hypercube is described in detail. The multicoding technique is a efficient, vectorized and memory saving scheme. Ring decomposition and the gather-scatter make the parallel version flexible and efficient. The performance is very good. Efficiency is over 90%. This parallel technique can be applied to a general class of not-so-memory-restrainted problems. This parallel code on MarkIIIfp outperforms the conventional supercomputers.

We thank M. Cross, G. Fox and P. Weichman for valuable discussions. This work is supported by DOE DE- GF03- 85ER 25009 and NSF DMR-87 15474. MSM thanks the Shell Foundation for a fellowship.

Table 1. Performance of MarkIIIfp for the Quantum Spin program. The timing (in seconds) for update 20 sweeps and 1 measurement, the speedup and the efficiency.

Size	Node	32	16	8	4	2	1
128×128	time	20.7	38.1	74.1	145.4	298	551
128×128	speedup	26.6	14.5	7.44	3.79	1.85	1
128×128	efficiency	.832	.904	.930	.948	.925	1
96×96	time		21.3	41.3	80.2	160	310
96×96	speedup		14.5	7.50	3.86	1.94	1
96×96	efficiency		.909	.937	.965	.968	1
64×64	time		9.86	18.4	35.5	69.8	139
64×64	speedup		14.1	7.55	3.91	1.99	1
64×64	efficiency		.881	.944	.979	.996	1

References and Footnotes

- [1] G.C.Fox, Concurrency: Practice and Experience, 1, 63 (1989).
- [2] See, for example, T. Prince and S. Anderson, Caltech report, for discovery of pulsars; P. Hipes, C. Winstead, V. McKoy, this proceeding, on the electron scattering; J. Salman, P. Quinn and M. Warren, Caltech report, on galaxies collision; and H.-Q. Ding, this proceeding, for the QCD calculations.
- [3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, S. Salman and D. Walker, Solving Problems on Concurrent Processors Vol. 1, Prentice Hall, Englewood Cliffs, New Jersey, 1988.
- [4] J. Tuazon et al, "Mark IIIfp Hypercube Concurrent Processor Architecture", p. 71; P. Burns et al "The JPL/Caltech Mark IIIfp Hypercube", p. 872, in Proceedings of The Third

- Conference on Hypercube Concurrent Computers and Applications, G. C. Fox, ed., ACM Press, New York, 1988.
- [5] J.G. Bednorz and K.A. Müller, Z.Phys. B64, 189 (1986); M.K.Wu et al, Phys.Rev. Lett. 58, 908 (1987).
- [6] H.-Q. Ding and M.S. Makivic, Phys.Rev.Lett, bf 64, 1449 (1990); M.S. Makivic and H.-Q. Ding, submitted to Phys. Rev.B.
- [7] H.-Q. Ding and M.S. Makivic, Caltech Report C³P-851, submitted to Physical Review Letters.
- [8] G. Fox, and S. Otto, *Physics Today*, **50** (May 1984).
- [9] H.-Q. Ding, Caltech report C3P-629, unpublished.
- [10] G. Chen, H.-Q. Ding and W. Goddard, in preparation.

Lattice QCD: Commercial vs. Home-grown Parallel Computers

Clive F. Baillie

Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, CA 91125, USA

Abstract

Numerical simulations of Lattice QCD have been performed on practically every computer, since its inception almost twenty years ago. Lattice QCD is an ideal problem for parallel machines as it can be easily domain decomposed. In fact, the urge to simulate QCD has led to the development of several home-grown parallel "QCD machines", in particular the Caltech Cosmic Cube, the Columbia Machine, IBM's GF11, APE in Rome and the Fermilab Machine. These machines were built because, at the time, there were no commercial parallel computers fast enough. Today however the situation has changed with the advent of computers like the Connection Machine 2 and the Ncube 2. Herein, I shall explain why Lattice QCD is such a parallel problem and compare two large-scale simulations of it one on the commercial Connection Machine and the other on the latest Caltech/JPL hypercube.

1. Introduction

Quantum Chromo-dynamics (QCD) simulations are consuming vast amounts of computer time these days, and promise to do so for at least the foreseeable future. The background for these calculations is two decades of great progress in our understanding of the basic particles and forces. Over time, the particle physics community has developed an elegant and satisfying theory which is believed to describe all the particles and forces which can be produced in today's high energy accelerators. The basic components of the so-called "Standard Model" are matter particles (quarks and leptons), and the forces through which they interact (electromagnetic, weak and strong). The electromagnetic force is the most familiar, and also the first to be understood in detail. The weak force is less familiar, but manifests itself in processes such as nuclear beta-decay,

for example. This piece of the Standard Model is now called the electroweak sector. The third part of the Standard Model is the QCD, the theory of the strong force, which binds quarks together into "hadrons", such as protons, neutrons, pions, and a host of other particles. The strong force is also responsible for the fact that protons and neutrons bind together to form the atomic nucleus. Currently we know of five types of quark (referred to as "flavors"): up, down, strange, charm and bottom; and expect at least one more (top) to show up soon. In addition to having a "flavor", quarks can carry one of three possible charges known as "color" (this has nothing to do with color in the macroscopic world!), hence Quantum Chromo-dynamics. The strong "color" force is mediated by particles called gluons, just as photons mediate light in electromagnetism. Unlike photons, though, gluons themselves carry a "color" charge and therefore interact with one another. This means that QCD is an extremely nonlinear theory which cannot be solved analytically. Hence we resort to numerical simulations.

2. Lattice QCD

To put QCD on a computer we proceed as follows. The four-dimensional space-time continuum is replaced by a four-dimensional hypercubic periodic lattice, of size $N = N_s \times N_s \times N_t$ with the quarks living on the sites and the gluons living on the links of the lattice. N_s is the spatial and N_t is the temporal extent of the lattice. The gluons are represented by 3×3 complex SU(3) matrices associated with each link in the lattice. This link matrix describes how the "color" of a quark changes as it moves from one site to the next. The action functional for the purely gluonic part of QCD is

$$S_G = \beta \sum_{P} (1 - \frac{1}{3} ReTr U_P), \qquad (1)$$

where

$$U_P = U_{i,\mu} U_{i+\mu,\nu} U_{i+\nu,\mu}^{\dagger} U_{i,\nu}^{\dagger} \tag{2}$$

is the product of link matrices around an elementary square or plaquette on the lattice – see Figure 1. Essentially all of the time in QCD simulations of gluons only is spent multiplying these SU(3) matrices together. The code for this, shown in the Appendix, reveals that its main component is the $a \times b + c$ kernel (which most supercomputers can do very efficiently). The partition function for full lattice QCD including quarks is

$$Z = \int D\psi D\bar{\psi}DU \exp(-S_G - \bar{\psi}(\not D + m)\psi), \quad (3)$$

where D + m is a large sparse matrix the size of the lattice squared. Unfortunately, since the quark variables ψ are anticommuting Grassmann numbers, there is no simple representation for them on the computer. Instead they must be integrated out, leaving a highly non-local fermion determinant:

$$Z = \int DU \det(\not D + m) \exp(-S_G), \qquad (4)$$

This is the basic integral one wants to evaluate numerically.

Note that the lattice is a mathematical construct used to solve the theory—at the end of the day, the lattice spacing a must be taken to zero to get back to the continuum limit. The lattice spacing itself does not show up explicitly in the partition function Z above. Instead the parameter $\beta = 6/g^2$, which plays the role of an inverse temperature, ends up controlling the lattice spacing $a(\beta)$. To take the continuum limit $a \rightarrow 0$ of lattice QCD one tunes $g \to 0$, or $\beta \to \infty$. Typical values used in simulations these days range from $\beta = 5.3$ to $\beta = 6.0$. This corresponds to $a \approx .1$ Fermi = 10^{-16} meter. Thus at current values of β a lattice with $N_{\bullet} = 20$ will correspond to a physical box about 2 Fermi on an edge, which is large enough to hold one proton without crushing it too much in the finite volume. Still the spacing a = .1 Fermi is not fine enough that we are close to the continuum limit. One can estimate that we still need to shrink the lattice spacing by something like a factor of 4, leading to an increase of a factor 44 in the number of points in the lattice in order to keep the box the same physical volume.

The biggest stumbling block preventing a large increase in the number of lattice points is the presence of the determinant $det(\not D+m)$ in the partition function. Physically, this determinant arises from closed quark loops. The simplest way to proceed is to ignore these quark loops and work in the

so-called "quenched" or "pure gauge" approximation. The quenched approximation assumes that the whole effect of quarks on gluons can be absorbed in a redefinition of the gluon interaction strength. Operationally, one generates gluon field configurations using only the pure gauge part of the action, and then computes the observables of interest in those backgrounds. For some quantities this may be a reasonable approximation. It is certainly orders of magnitude cheaper, and for this reason, most all simulations to date have been done using it. To investigate the fully realistic theory, though, one has to go beyond the quenched approximation and tackle the fermion determinant.

There have been many proposals for dealing with the determinant. The first algorithms tried to compute the change in the determinant when a single link variable was updated. This turned out to be prohibitively expensive. Today, the preferred approach is the so-called "Hybrid Monte Carlo" algorithm [1]. The basic idea is to invent some dynamics for the variables in the system in order to evolve the whole system forward in (simulation) time and then do a Metropolis accept/reject for the entire evolution on the basis of the total energy change. The great advantage is that the whole system is updated at one fell swoop. The disadvantage is that if the dynamics is not correct then the acceptance will be very small. Fortunately (and this one of very few fortuitous happenings where fermions are concerned) good dynamics can be found: the Hybrid algorithm [2]. This is a neat combination of the deterministic microcanonical method [3] and the stochastic Langevin method [4]. which yields a quickly-evolving, ergodic algorithm for both gauge fields and fermions. The computational kernel of this algorithm is the repeated solution of systems of equations of the form

$$(\not\!\!D+m)\phi=\eta, \qquad \qquad (5)$$

where ϕ and η are vectors which live on the sites of the lattice. To solve these equations one typically uses conjugate gradient or one of its cousins, since the fermion matrix $(\not D+m)$ is sparse. For more details, see [5]. Such iterative matrix algorithms have as their basic component the $a \times b + c$ kernel, so again computers which do this efficiently will run QCD both with and without fermions well.

However one generates the gauge configurations U, using the quenched approximation or not, one then has to compute the observables of interest. For observables involving quarks one runs into expressions like $(\psi(x)\bar{\psi}(y))$ involving pairs of quark fields at different points. Again because of the Grassmann

nature of fermions fields, one has to express this quantity as

$$\langle \psi(x)\bar{\psi}(y)\rangle = (\not\!\!D + m)_{xy}^{-1}. \tag{6}$$

And again one computes as many columns of the inverse as needed by solving systems equations like (5) above. For simulations of full QCD with quark loops, this phase of the calculation is a small overhead, while for quenched calculations it is the dominant part. So whether quenched or not, most of the computer time is spent in applying conjugate gradient to solve large systems of linear equations.

3. Home-grown QCD Machines

Today the biggest resources of computer time for research are the conventional supercomputers at the NSF and DOE centers. The centers are continually expanding their support for lattice gauge theory, but it may not be long before they are overtaken by several dedicated efforts involving concurrent computers. It is a revealing fact that the development of most high performance parallel computers—the Caltech Cosmic Cube, the Columbia Machine, IBM's GF11, APE in Rome, the Fermilab Machine—was actually motivated by the desire to simulate lattice QCD.

Geoffrey Fox and Chuck Seitz at Caltech built the first hypercube computer, the Cosmic Cube or Mark I, in 1983 [6]. It had 64 nodes, each of which was an Intel 8086/87 microprocessor with 128 KB of memory, giving a total of about 2 Mflops (measured for QCD). This was quickly upgraded to the Mark II hypercube with faster chips, twice the memory per node and twice the number of nodes in 1984 [7] . Now QCD is running at 600 Mflops sustained on the latest Caltech hypercube: the 128-node Mark IIIfp (built by JPL) [8]. Each node of the Mark IIIfp hypercube contains two Motorola 68020 microprocessors, one for communication and the other for calculation, with the latter supplemented by one 68881 coprocessor and a 32-bit Weitek floating point processor.

Norman Christ and Tony Terrano at Columbia built their first parallel computer for doing lattice QCD calculations in 1984 [9]. It had 16 nodes, each of which was an Intel 80286/87 microprocessor plus a TRW 22-bit floating point processor with 1 MB of memory, giving a total peak performance of 256 Mflops. This was improved in 1987 using Weitek rather than TRW chips so that 64 nodes give 1 Gflops peak [10]. Very recently, Columbia have finished building their third machine: a 256-node 16 Gflops lattice QCD computer [11].

Don Weingarten at IBM has been building the GF11 since 1984—it is expected he will start running in production in 1990 [12]. The GF11 is an SIMD machine comprising 576 Weitek floating point processors, each performing at 20 Mflops to give the total 11 Gflops peak implied by the name.

The APE (Array Processor with Emulator) computer is basically a collection of 3081/E processors (which were developed by CERN and SLAC for use in high energy experimental physics) with Weitek floating point processors attached [13] . However, these floating point processors are attached in a special way-each node has four multipliers and four adders in order to optimize complex $a \times b + c$ calculations, which form the major component of all lattice QCD programs. This means that each node has a peak performance of 64 Mflops. The first, small machine—Apetto—was completed in 1986 and had 4 nodes yielding a peak performance of 256 Mflops. Currently, they have a second generation of this with 1 Gflops peak from 16 nodes. By 1992, the APE collaboration hopes to have completed the 100 Gflops 4096-node "Apecento" [14].

Not to be outdone, Fermilab is also using its high energy experimental physics emulators in constructing a lattice QCD machine for 1991 with 256 of them arranged as a 2⁵ hypercube of crates, with 8 nodes communicating through a crossbar in each crate [15]. Altogether they expect to get 5 Gflops peak performance.

These performance figures are summarized in Table 1. The "real" performances are the actual performances obtained on QCD codes; in Figure 2 we plot these as a function of the year the QCD machines started to produce physics results. The surprising fact is that the rate of increase is very close to exponential, yielding a factor of ten every two years! On the same plot we show our estimate of the computer power needed to redo this year's quenched calculations on a 1284 lattice. This estimate is also a function of time, due to algorithm improvements. Extrapolating both lines, we see the outlook for lattice QCD is rather bright. Reasonable results for the "harder" physical observables should be available within the quenched approximation in the mid-90's. With the same computer power we will be able to redo today's quenched calculations using dynamical fermions (but still on today's size of lattice). This will tell us how reliable the quenched approximation is. Finally, results for the full theory with dynamical fermions on a 1284 lattice should follow early in the next century (!), when computers are two or three orders of magnitude more powerful again.

Table 1
Peak and real performances in Mflops of "homebrew" QCD machines

Computer	Year	Peak	Real
Caltech I	1983	3	2
Caltech II	1984	9	6
Caltech III	1989	2000	600
Columbia I	1984	256	20
Columbia II	1987	1000	200
Columbia III	1990	16000	6000
IBM GF11	1990	11000	10000*
APE I	1986	256	20
APE II	1988	1000	200
APE III	1992	100000	20000*
Fermilab	1991	5000	1200*

* All real times are measured except these predicted ones.

With this brief review in hand, we now turn to a comparison of QCD running on one home-grown computer – the Caltech/JPL Mark IIIfp hypercube – with the commercially available TMC Connection Machine 2.

4. QCD on the Caltech/JPL Mark IIIfp

Decomposing QCD onto a d-dimensional hypercube distributed memory computer (with 2^d nodes) is particularly simple. One takes the N = $M2^d$ lattice and splits it up into 2^d sublattices, each containing M sites, one of which is placed in each node. Due to the locality of the action, eq. (2), it is possible to assign the sublattices so that each node needs only to communicate with others to which it is directly connected in hardware. As a result of this fact the characteristic timescale of the communication, t_{comm} , is minimal and corresponds to roughly the time taken to transfer a single SU(3) matrix from one node to its neighbor. Conversely we can characterize the calculational part of the algorithm by a timescale, t_{calc} , which is roughly the time taken to multiply together two SU(3) matrices. For all hypercubes built without floating point accelerator chips $t_{comm} \ll t_{calc}$ and hence QCD simulations are extremely "efficient", where efficiency is defined by the relation

$$\epsilon = \frac{T_1}{kT_1},\tag{7}$$

where T_k is the time taken for k processors to perform the given calculation. Typically such calculations have efficiencies in the range $\epsilon \geq .90$ which

means they are ideally suited to this type of computation since doubling the number of processors approximately halves the total computational time required for solution. However, as we shall see, the picture changes dramatically when fast floating point chips are used; then $t_{comm} \simeq t_{calc}$ and one must take some care in coding to obtain maximum performance.

QCD simulations have been done on all the Caltech hypercubes; the most recent being a high statistics, large lattice study of the string tension in pure gauge QCD on the Mark IIIfp [16]. For this the 128-node hypercube performs at 0.6 Gflops. As each node runs at 6 Mflops this corresponds to a speedup of 100, and hence an efficiency of 78%. These figures are for the most highly optimized code. The original version of the code written in C ran on the Motorola chips at 0.085 Mflops and on the Weitek chips at 1.3 Mflops. The communication time, which is roughly the same for both, is less than a 2% overhead for the former but nearly 30% for the latter. When the computationally intensive parts of the calculation are written in assembly code for the Weitek this overhead becomes almost 50%. This 0.9 msec of communication, shown in lines 2 and 3 in Table 2, is dominated by the hardware/software message startup overhead (latency), because for the Mark II-If p the node to node communication time, t_{comm} , is given by

$$t_{comm} \simeq (150 + 2 * W) \quad \mu sec,$$

where W is the number of words transmitted. To speed up the communication we update all even (or odd) links (8 in our case) in each node, allowing us to transfer 8 matrix products at a time, instead of just sending one in each message. This reduces the 0.9 msec by a factor of

$$\frac{8*(150+18*2)}{150+8*18*2}=3.4$$

to 0.26 msec. On all hypercubes with fast floating point chips – and on most hypercubes without for less computationally intensive codes – such vectorization of communication is often important. In Figure 3, the speedups for many different total lattice sizes are shown. For the largest lattice size, the speedup is 100 on the 128-node. The speedup is almost linear in number of nodes. As the total lattice volume increases, the speedup increases, because the ratio of calculation/communication increases. For more information on this performance analysis, see [17].

5. QCD on the TMC Connection Machine 2

The Connection Machine is also very well suited for large-scale simulations of QCD. The CM-2 is a distributed-memory, Single-Instruction Multiple-Data (SIMD) massively-parallel processor comprising up to 65536 (64K) processors [18]. Each processor consists of an arithmetic-logic unit (ALU), 8 or 32 Kbytes of random-access memory (RAM) and a router interface to perform communications among the processors. There are sixteen processors and a router per custom VLSI chip, with the chips being interconnected as a twelve-dimensional hypercube. Communications among processors within a chip work essentially like a cross-bar interconnect. The router can do general communications but we require only local ones for QCD so we use the fast nearest-neighbor communication software called NEWS. The processors deal with one bit at a time, therefore the ALU can compute any two boolean functions as output from three inputs, and all data paths are 1-bit wide. In the current version of the Connection Machine (the CM-2) groups of 32 processors (two chips) share a 32-bit (or 64-bit) Weitek floating point chip, and a transposer chip which changes 32 bits stored bit-serially within 32 processors into 32 32-bit words for the Weitek, and vice versa.

The high-level languages on the CM, such as *Lisp and CM-Fortran, compile into an assembly language called Paris (Parallel Instruction Set). Paris regards the 64K bit-serial processors as the fundamental units in the machine, and so well represents the global aspects of the hardware. However, floating point computations are not very efficient in the Paris model. This is because in Paris 32-bit floating point numbers are stored "field-wise", that is, successive bits of the word are stored at successive memory locations of each processors memory. However, 32 processors share one Weitek chip which deals with words stored "slice-wise", that is, stored across the processors, one bit in each. Therefore to do a floating point operation, Paris loads in the field-wise operands, transposes them slice-wise for the Weitek (using the transposer chip), does the operation and transposes the slice-wise result back to field-wise for memory storage. Moreover, every operation in Paris is an atomic process, that is, two operands are brought from memory and one result is stored back to memory so no use is made of the Weitek registers for intermediate results. Hence, to improve the performance of the Weiteks, a new assembly language called CMIS (CM Instruction Set) has been written, which models the local architectural features much better. In fact, CMIS ignores the bit-serial processors and thinks of the machine in terms of the Weitek chips. Thus data can be stored slice-wise, eliminating all the transposing back and forth. CMIS allows effective use of the Weitek registers, creating a memory hierarchy, which combined with the internal buses of the Weiteks offers increased bandwidth for data motion.

Currently, the Connection Machine is the most powerful commercial QCD machine available: the "Los Alamos collaboration" is running full QCD at a sustained rate of almost 2 Gflops on a 64K CM-2 [19] . As was the case for the Mark IIIfp hypercube, in order to obtain this performance one must resort to writing assembly code for the Weitek chips and for the communication. Our original code, written entirely in *Lisp, achieved around 1 Gflops. As shown in Table 3, this code spends 34% doing communication. When we rewrote the most computationally intensive part in the assembly language CMIS, this rose to 54%. In order to obtain maximum performance we are now rewriting the communication part of our code to make use of "multi-wire NEWS" which will allow us to communicate in all 8 directions on the lattice simultaneously thereby reducing the communication time by a factor of 8 and speeding up the code by another factor of 2.

6. Conclusions

It is interesting to note that when the various groups began building their "homebrew" QCD machines it was clear that they would out-perform all commercial (traditional) supercomputers; however, now that commercial parallel supercomputers have come of age [20] the situation is not so obvious.

On the original versions of both commercial and home-grown parallel computers (without fast floating point chips) one could get good performance from one's favorite high-level language. Now, however, as most of these machines do have fast floating point hardware, one must resort to lover-level assembly programming to obtain maximum performance. Having done just that, we are running QCD at 0.6 Gflops on the Caltech/JPL Mark IIIfp hypercube and at 1.65 Gflops on the TMC Connection Machine 2.

References

- S. Duane, A.D. Kennedy, B.J. Pendleton and D. Roweth, Hybrid Monte Carlo, *Phys. Lett.* B195, 216 (1987).
- [2] S. Duane, Stochastic Quantization versus the Microcanonical Ensemble: Getting the best of both worlds, Nucl. Phys. B257, 652 (1985).
- [3] D. Callaway and A. Rahman, Lattice Gauge Theory in the Microcanonical Ensemble, Phys. Rev. D28, 1506 (1983); J. Poloyni and H.W. Wyld, Microcanonical Simulation of Fermionic Systems, Phys. Rev. Lett. 51, 2257 (1983).
- [4] G. Parisi and Y. Wu, Perturbation Theory without Gauge Fixing, Sci. Sin. 24, 483 (1981); G.G. Batrouni, G.R. Katz, A.S. Kronfeld, G.P. Lepage, B. Svetitsky and K.G. Wilson, Langevin Simulations of Lattice Field Theories, Phys. Rev. D32, 2736 (1985).
- [5] R. Gupta, G.W. Kilcup and S.R. Sharpe, Tuning the Hybrid Monte Carlo Algorithm, *Phys. Rev.* D38, 1278 (1988).
- [6] C. L. Seitz, The Cosmic Cube, Comm. of the ACM 28, 22 (1985).
- [7] J. Tuazon et al, The Caltech/JPL Mark II hypercube concurrent processor, In IEEE 1985 Conference on Parallel Processing, St. Charles, Illinois, August 1985.
- [8] J. C. Peterson et al, The Mark III hypercube ensemble concurrent computer, In IEEE 1985 Conference on Parallel Processing, St. Charles, Illinois, August 1985.
- [9] N. H. Christ and A. E. Terrano, A Very Fast Parallel Processor, *IEEE Trans. on Computers* C-33, 344 (1984).
- [10] F. Butler, Status of the Columbia Parallel Processors, Nucl. Phys. B (Proc. Suppl.) 9, 557 (1989).
- [11] N. H. Christ, Status of the Columbia 256node Parallel Supercomputer, In Proc. of the Int. Symp. Lattice 89, Capri, September 1989, to appear in Nucl. Phys B (Proc. Suppl.) (1990).
- [12] J. Beetem, M. Denneau and D. Weingarten, The GF11 Supercomputer, In IEEE Proc. of the 12th Annual Int. Symp. on Computer Architecture, IEEE Computer Society, Washington D.C., 1985; D. Weingarten, Progress Report on the GF11 Project, In Proc. of the Int. Symp. Lattice 89, Capri, September 1989, to appear in Nucl. Phys. B (Proc. Suppl.) (1990).
- [13] M. Albanese et al, The APE Computer: An Array Processor Optimized for Lattice Gauge

- Theory Simulations, Comp. Phys. Commun. 45, 345 (1987).
- [14] R. Tripiccione, Machines for Theoretical Physics, In Proc. of the Int. Symp. Lattice 89, Capri, September 1989, to appear in Nucl. Phys. B (Proc. Suppl.) (1990).
- [15] T. Nash et al, The Fermilab Advanced Computer Program Multi-Processor Project, In Proc. Conf. Computing in High Energy Physics, Amsterdam, June 1985 (North-Holland, Amsterdam, 1986); M. Fischler, Hardware and System Software Requirements for Algorithm Development, In Proc. of the Int. Symp. Lattice 89, Capri, September 1989, to appear in Nucl. Phys. B (Proc. Suppl.) (1990).
- [16] H.-Q. Ding, C. F. Baillie and G. C. Fox, Calculation of the heavy quark potential at large separation on a hypercube parallel computer, to appear in Phys. Rev. D. (1990).
- [17] H.-Q. Ding, The Mark IIIfp Hypercube: Performance of a QCD code, Caltech preprint C3P-799, submitted to Comp. Phys. Commun. (1990).
- [18] W. D. Hillis, The Connection Machine (MIT Press, Cambridge, MA, 1985); Thinking Machines Corporation, Connection Machine Model CM-2 Technical Summary, TMC Technical Report HA87-4, Cambridge, MA, 1987.
- [19] C. F. Baillie, R. G. Brickner, R. Gupta and S. L. Johnsson, QCD with dynamical fermions on the Connection Machine, In *Proc. of Super*computing 89, Reno, Nevada (ACM Press, New York, 1989).
- [20] G. C. Fox, Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech, Concurrency: Practice and Experience 1, 1 (1989).

Table 2
Link update time (msec) on Mark IIIfp node for various levels of programming

Programming level	Calc. time	Comm. time	Total time	Mflops
Motorola MC68020/68881 in C	52	0.86	53	0.085
Weitek XL all in C	2.25	0.90	3 .15	1.4
Weitek XL parts in Assembly	0.94	0.90	1.84	2.4
Weitek XL Assembly, vec. comm.	0.94	0.26	1.20	3.8
Weitek XL Assembly, no comm.	0.94	0.0	0.94	4.8

Table 3
Fermion update time (sec) on 64K Connection Machine for various levels of programming

Programming level	Calc. time	Comm. time	Total time	Mflops
All in *Lisp Inner loop in CMIS	8.7 3.3	4.5 3.9	13.2 7.2	900 1650
Multi-wire CMIS†	< 3.3	0.5	< 3.8	> 3100

[†] projected numbers

Fig. 1.

Illustration of plaquette calculation

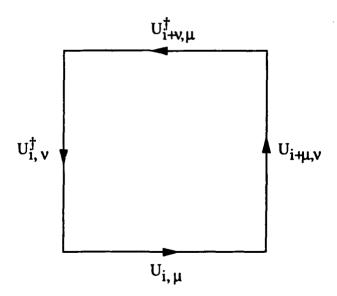


Fig. 2. Computational power of QCD machines

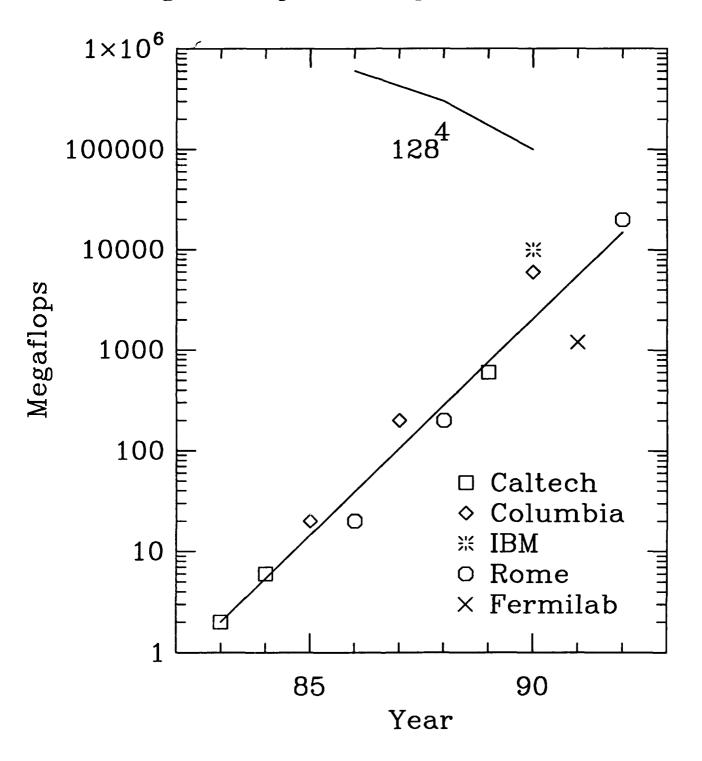
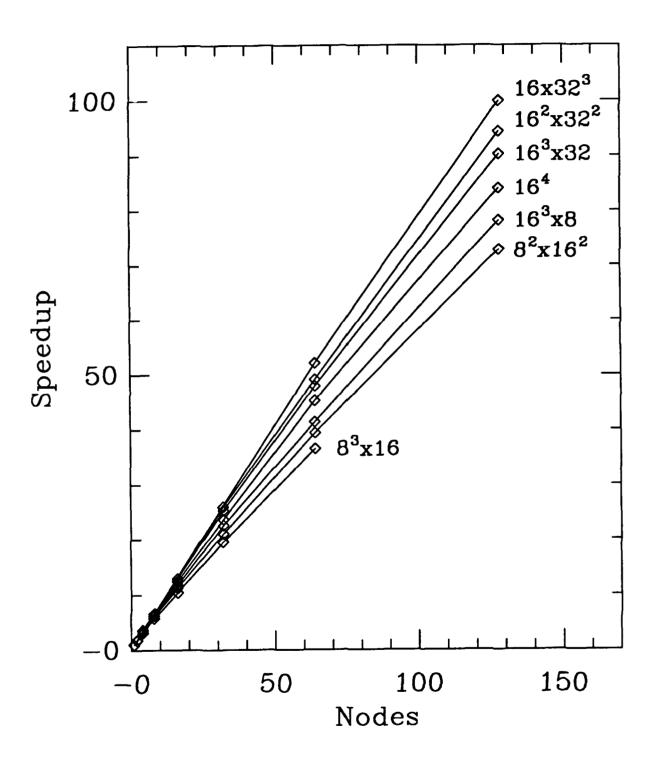
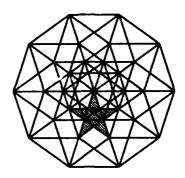


Fig. 3. Speedup of QCD code on Mark IIIfp





The Fifth Distributed Memory Computing Conference

16: Electromagnetic Scattering Problems

The Finite Element Solution of Two-Dimensional Transverse Magnetic Scattering Problems on the Connection Machine

Scott Hutchinson¹ Steven Castillo¹ Edward Hensel² Kim Dalton¹

¹Department of Electrical and Computer Engineering ²Department of Mechanical Engineering scastill@nmsu.edu Box 3-o

New Mexico State University, Las Cruces, New Mexico 88003

April 6, 1990

Abstract

A study is conducted of the finite element solution of the partial differential equations governing twodimensional electromagnetic field scattering problems on a SIMD computer. A nodal assembly technique is introduced which maps a single node to a single processor. The physical domain is first discretized in parallel to yield the node locations of an O-grid mesh. Next, the system of equations is assembled and then solved in parallel using a conjugate gradient algorithm for complexvalued, non-symmetric, non-positive definite systems. Using this technique and Thinking Machines Corporation's Connection Machine-2 (CM-2), problems with more than 250k nodes are solved.

Results of electromagnetic scattering, governed by the 2-d scalar Helmholtz wave equations are presented for a variety of infinite cylinders and airfoil crosssections. Solutions are demonstrated for a wide range of objects. A summary of performance data is given for the set of test problems.

1 Introduction

The finite element technique is a method which allows for the approximate solution of partial differential equations over some finite domain. Because partial differential equations govern various physical phenomena, the technique has applications in many disciplines. Here, a study is conducted of the finite element solution of the partial differential equations governing two-dimensional electromagnetic field scattering problems on a SIMD computer.

First, the weak form of the continuous governing equations are given. Second, the mapping of the finite element program onto Thinking Machines Corporation's Connection Machine using nodal assembly is described. Third, results are presented for a variety of scattering shapes. Lastly, conclusions are drawn and future research discussed.

2 Problem Formulation

The equations of interest are the 2-d scalar and vector Helmholtz wave equations [1]. The equations are applied over an open region artificially truncated with an absorbing boundary condition [2]. The scalar equation

$$\nabla \cdot \frac{1}{\mu_r} \nabla E_z + k_0^2 \epsilon_r E_z = 0 \tag{1}$$

governs the transverse magnetic (TM) normal incident case and the vector equation

$$\nabla \times \frac{1}{\epsilon_r} \nabla \times \mathbf{H} - k_0^2 \mu_r \mathbf{H} = 0$$
 (2)

governs the TM oblique incident case. H represents the unknown magnetic field and E_z represents the zcomponent of the unknown electric field. Each case can be written

$$E_z = E_z^i + E_z^s$$

$$\mathbf{H} = \mathbf{H}^i + \mathbf{H}^s$$
(3)

$$\mathbf{H} = \mathbf{H}^{\iota} + \mathbf{H}^{\prime} \tag{4}$$

where E_z^i and \mathbf{H}^i represent the known incident fields while E_z^s and H^s are the unknown scattered fields (Fig-

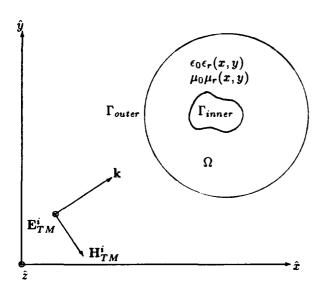


Figure 1: Open region scattering problem.

Applying the Galerkin technique, the scalar equation is written

$$\int_{\Omega} \left(\frac{1}{\mu_r} \nabla T \cdot \nabla E_z^s - k^2 \epsilon_r T E_z^s \right) d\Omega
- \int_{\Gamma} T \left(\alpha T E_z^s - \beta \frac{\partial T}{\partial \phi} \frac{\partial E_z^s}{\partial \phi} \right) d\Gamma
= \int_{\Omega} T \left[\nabla \cdot \left(\frac{1}{\mu_r} \nabla E_z^i \right) + k^2 \epsilon_r E_z^i \right] d\Omega$$
(5)

where the unknown is the scattered electric field and the vector equation is written

$$\int_{\Omega} \left(\frac{1}{\epsilon_r} \nabla \times \mathbf{T} \cdot \nabla \times \mathbf{H} - k^2 \mu_r \mathbf{T} \cdot \mathbf{H} \right) d\Omega$$

$$+ \int_{\Gamma} \left[\alpha \mathbf{T} \cdot \mathbf{H}_{tan} - \beta (\hat{r} \cdot \nabla \times \mathbf{T}) (\hat{r} \cdot \nabla \times \mathbf{H}) \right] d\Gamma$$

$$= \int_{\Gamma} \left[\alpha \mathbf{T} \cdot \mathbf{H}_{tan}^i - \beta (\hat{r} \cdot \nabla \times \mathbf{T}) (\hat{r} \cdot \nabla \times \mathbf{H}^i) \right] d\Gamma$$

$$- \int_{\Gamma} \mathbf{T} \cdot \hat{r} \times \nabla \times \mathbf{H}^i d\Gamma \quad (6)$$

where the unknown is the total magnetic field. In each case the Bayliss-Turkel radiation condition has been applied to satisfy the Neumann boundary condition on the outer boundary.

In order to obtain the final finite-element form, these equations are discretized and presented as a linear system of equations

$$\mathbf{K}\mathbf{u} = \mathbf{b} \tag{7}$$

for u the unknown.

3 Nodal Mapping

The SIMD computer used is Thinking Machines Corporation's Connection Machine 2 (CM-2). Briefly, the CM-2 is described as a SIMD (Sequential Instruction Multiple Data) or data parallel type of parallel computer. This means that each computer instruction operates on data stored in a processor array. Each processor in the array holds a single data item. The CM-2 may be configured to have up to 64k (k=1024) physical processors each with its own local memory. Computationally, each physical processor may be subdivided into some number of virtual processors where the computational resources of the physical processor are shared among its virtual processors. The virtual processor ratio, then, is the ratio of the number of virtual processors assigned to each physical processors and must equal an integer power of 2. For a more complete description, see [3].

While SIMD computers have been in existence for a number of years, finite element algorithms for them have been few. One reason is that, as with all parallel architectures, techniques which may be mature on serial computers must be altered and sometimes discarded in favor of more applicable algorithms. This paper introduces a new nodal basis mapping of the finite element algorithm onto the CM-2.

One difficulty with implementing finite element algorithms on a SIMD computer is the choice of the data item. To achieve a relatively high level of efficiency as well as to limit communication, a data item which may be maintained throughout the algorithm is desirable. Typically, finite element algorithms operate on an elemental level during the calculation of the system of equations and then assemble these elemental equations to a global set of equations which exist on the nodal level. This global set of equations is then solved to yield results at the nodal level. This may be seen as having two different data items during different portions of the program and previous implementations of this mapping on the CM-2 have proved inefficient [4], [5]. To avoid this inefficiency, an algorithm which uses a nodal level data set throughout the program has been developed for use on the CM-2. While the solution on the nodal level remains basically the same as previous finite element algorithms on the CM-2 [6], [7], the calculation of the system of equations is done on the nodal level using what has been termed nodal assembly.

3.1 Mesh Generation

The nodal-basis mapping assigns a node to a processor. This mapping is maintained throughout the program, from discretization through solution. During discretization, each processor calculates its position in the prob-

lem domain based on information which describes the domain geometry. Each processor also determines its boundary status. To enhance the speed of the program, a parallel O-grid mesh generator is used to generate meshes. The O-grid meshes allow the use of nearest neighbor (NEWS) communication grid while the parallel mesh generation means that only geometry data need be specified on a front-end preprocessor.

A mesh is generated by the set of points formed by the intersection of the lines of a boundary conforming curvilinear coordinate system. The problem of interest is a two-dimensional, multiply-connected, arbitrary region with specified inner and outer boundaries. The boundary values are specified in cartesian coordinates (x, y) and are transformed to curvilinear coordinates (s,t). In the transformed region, algebraic interpolation is used to generate the physical cartesian coordinates (x, y). See [8] for a complete description.

3.2 Nodal Assembly

The nodal assembly technique makes use of the concept of a nodal region which contains a given node and its neighboring nodes and elements as in Figure 2. Each processor simply calculates the local interaction coefficients associated with its row in the global system of equations as well as the forcing value. Since the interactions are local, nearest-neighbor communication is used. This portion of the algorithm is somewhat inefficient in applying boundary conditions since only processors which represent boundary nodes are active during this phase of the program. However, this may only be slightly detrimental to the overall efficiency of the program if the boundary-condition calculations are not too laborious.

System Solution 3.3

Once calculated, the system of equations is solved using a conjugate-gradient based algorithm [9]. Conjugate gradient algorithms have been used previously on the CM-2 for the solution of linear systems [6], [7]. This is because they are a collection of various matrix and vector operations which can be performed with a high level of concurrency. Further, in the case of a regular grid, all the system coefficients represent local interactions and so any interprocessor communication will be nearest neighbor. Thus, communication is also optimized using this solution technique. However, in contrast with previous finite element algorithms on the CM-2, the conjugate gradient algorithm used here is one which must handle a complex-valued, non-positive definite system of equations. It is given as

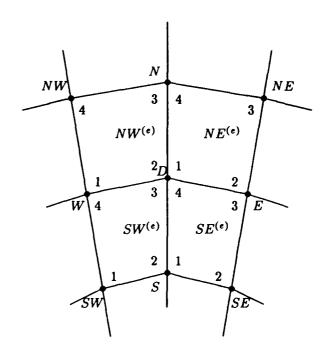


Figure 2: Nodal Region of node "D" indicating nearest neighbor nodes and adjacent elements.

Initialize:

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{K}\mathbf{u}_0 \tag{8}$$

$$\mathbf{p}_0 = \mathbf{K}^T \mathbf{r}_0 \tag{9}$$

Iterate:

$$a_i = \frac{|\mathbf{K}^T \mathbf{r}_i|^2}{|\mathbf{K} \mathbf{p}_i|^2} \tag{10}$$

$$\mathbf{u}_{i+1} = \mathbf{u}_i + a_i \mathbf{p}_i \tag{11}$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i - a_i \mathbf{K} \mathbf{p}_i \tag{12}$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i - a_i \mathbf{K} \mathbf{p}_i$$

$$b_i = \frac{|\mathbf{K}^T \mathbf{r}_{i+1}|^2}{|\mathbf{K}^T \mathbf{r}_i|^2}$$
(12)

$$\mathbf{p}_{i+1} = \mathbf{K}^T \mathbf{r}_{i+1} + b_i \mathbf{p}_i \tag{14}$$

where the choice of uo is arbitrary. Note here that the matrix-transpose implies the conjugate-transpose.

Figure 3 is a flow chart of the finite element program as implemented on the CM-2.

Results

The method described above has been implemented on the CM-2 using the C-Paris (PARallel Instruction Set) programming protocol for the Connection Machine [10]. This program was used to obtain results for the solution of electromagnetic wave scattering from a variety

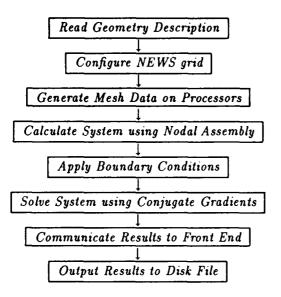


Figure 3: Flow chart for CM-2 nodal-basis finite element program.

of 2-dimensional objects. Table 1 gives some test problems for scattering from perfect electric conducting objects. The first 4 cases are all cylindrical shapes for which a semi-analytical solution is available for accuracy verification. The last case is an airfoil with NACA number 0010. All floating point calculations are done using 32 bit arithmetic and the floating-point acceleration hardware available on the CM-2. The conjugate gradient algorithm was halted when the following was satisfied

$$\frac{|\mathbf{r_i}|}{|\mathbf{b}|} < 10^{-4} \tag{15}$$

Figures 4-13 represent magnitude and phase plots of the fields for the cases listed in Table 1. In each case, the incident plane wave is taken as traveling in the x-direction. The total field magnitude becomes zero on the boundary and displays a shadow region behind the conducting body. The phase plots illustrate that lines of constant phase approach the perfect conducting inner boundary at normal incidence. This follows from the boundary condition that

$$E_{tan} = 0$$

on the boundary.

Table 2 gives timing and Megaflop ratings achieved on the same problems. All timing results were obtained using the CM timing facility. As Table 2 illustrates, projected floating point computations from 200-400 MFlops have been achieved during both phases of the algorithm. Further, the MFlop ratings extrapolated to the same virtual processor ratio run on a full 64k CM-2

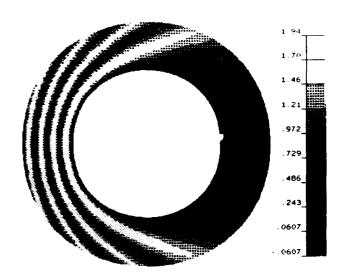


Figure 4: Case 1 magnitude: Total field magnitude for scattering from a perfect electric conducting cylinder with $a = 3\lambda$ and $b = 5\lambda$

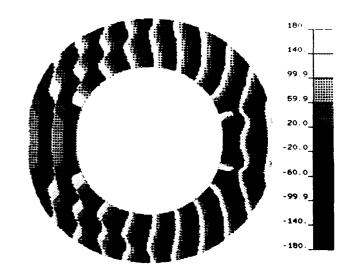


Figure 5: Case 1 phase: Total field phase for scattering from a perfect electric conducting cylinder with $a=3\lambda$ and $b=5\lambda$

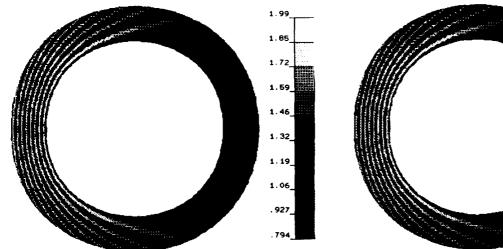
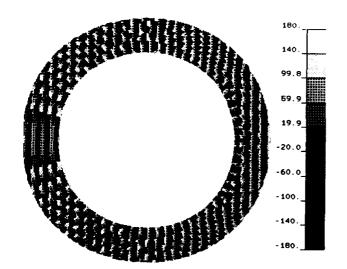
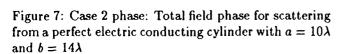


Figure 6: Case 2 magnitude: Total field magnitude for scattering from a perfect electric conducting cylinder with $a=10\lambda$ and $b=14\lambda$

1. 99 1. 85 1. 72 1. 69 1. 46 1. 32 1. 19 1. 06 . 926 . 794

Figure 8: Case 3 magnitude: Total field magnitude for scattering from a perfect electric conducting cylinder with $a=10\lambda$ and $b=14\lambda$. Twice the nodal density was used in both the radial and circumferential directions as for Case 2





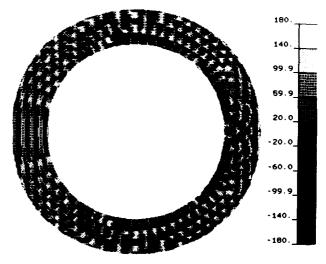


Figure 9: Case 3 phase: Total field phase for scattering from a perfect electric conducting cylinder with $a = 10\lambda$ and $b = 14\lambda$. Twice the nodal density was used in both the radial and circumferential directions as for Case 2

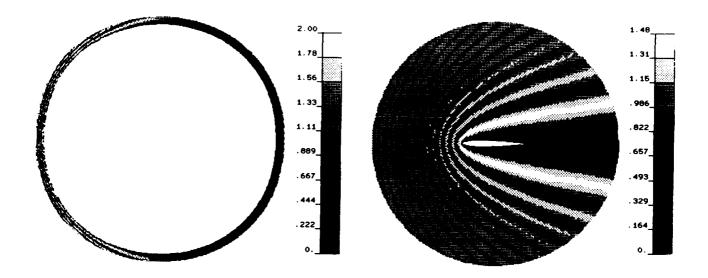


Figure 10: Case 4 magnitude: Total field magnitude for scattering from a perfect electric conducting cylinder with $a=30\lambda$ and $b=32\lambda$

Figure 12: Airfoil magnitude: Total field magnitude for scattering from a perfect electric conducting airfoil with chord length = 5λ and $b = 9.5\lambda$. NACA number is 0010

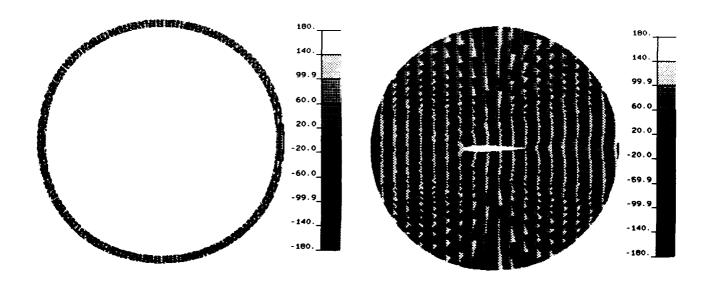


Figure 11: Case 4 phase: Total field phase for scattering from a perfect electric conducting cylinder with $a=30\lambda$ and $b=32\lambda$

Figure 13: Airfoil phase: Total field phase for scattering from a perfect electric conducting airfoil with chord length $= 5\lambda$ and $b = 9.5\lambda$. NACA number is 0010

show capabilities on the order of 1.5 GFlops for both portions. The finite element mapping described above will allow the solution of problems in excess of 4 million nodes on a fully configured CM-2 with the larger 256-kbit memory chips. Because of this capability, objects of electrical sizes (dimension in terms of wavelengths) exceeding 100 wavelengths may studied using the finite element method. This has not previously been possible.

Figure 14 shows a plot of the "relative speedup" demonstrated by the Fill and Solve portions of the program. This "relative speedup" is defined as

$$rsp \equiv \frac{t_{minp}}{t_{np}} \tag{16}$$

where t_{minp} is the execution time of a given problem on the minimum number of processors possible (highest virtual processor ratio) and t_{np} is the execution time on some number of processors. Note that the graph illustrates the speedup over the largest possible range of these ratios for this program using a CM-2 with processor memories of 64k-bits.

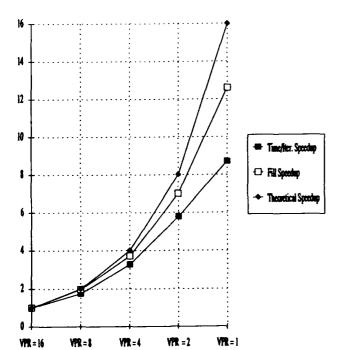


Figure 14: Relative Speedup.

Table 1: Test cases.

]]	Num. Nodes			
Case	a	b	Circ.	Radial	Total	Ratio	
1	3 λ	5 λ	512	32	16384	1	
2	10 λ	14 λ	1024	64	65536	4	
3	10 λ	14 λ	2048	128	262144	16	
4	30 λ	32 λ	2048	32	65536	4	
5	5 λ*	9.5 λ	1024	128	131072	8	

* - the chord length of the airfoil was taken as 5λ .

Table 2: Timings and Mflop ratings for the above cases.

		Time (s	MFlops		
Case	Fill	Solve	Total	Fill	Solve
1	0.05	33.52	113.25	365	211
2	0.17	137.88	314.65	430	322
3	0.78	1811.82	1967.79	375	399
4	0.17	125.24	224.72	430	318
5	0.32	-	6798	457	-

Table 3: Mflop ratings extrapolated to the virtual processor ratio implemented on a full 64k processor CM-2.

VP	Projected MFlops				
Ratio	Fill	Solve			
1	1461	846			
4	1719	1289			
16	1500	1596			

5 Conclusions and Future Research

Using the nodal-assembly technique, a finite element program is implemented on a data parallel computer in a manner which allows the use of the same data structure throughout the program, from discretization through solution. Nodal-assembly mapping provides for a relatively efficient program.

From the work presented here, conclusions may be drawn.

- Nodal assembly ailows the mapping of one finite element node onto one virtual processor. This mapping is maintained throughout the program.
- Using first order quadrilaterals and a regular mesh, the mapping may be configured in a NEWS grid, allowing nearest-neighbor communication.
- The mapping described above will permit a maximum virtual processor ratio of 16 under the current CM-2 memory limitations (64k bits per processor). On a 64k processor machine, this allows a maximum of 1048576 nodes.
- Nodal assembly is inefficient when handling boundary conditions. This is because only processors on a given boundary are active during this portion of a program.
- Nodal basis mapping is well suited for use with a conjugate-gradient iterative solution. All the matrix and vector operations can be computed with a high level of concurrency. Nearest neighbor communication is again used in performing the matrixvector products.

With respect to the CM-2, several things need be said. First, although all these examples were run on a machine with only a 32 bit floating point accelerator, 64 bit accelerators are now available to allow double precision floating point calculations in hardware. Second, the individual processor memory has been increased from 64k bits to 256k bits on some machines. This would effectively allow the solution of problems with 4 times the number of nodes. The 64k bits of processor memory allowed for a maximum virtual processor ratio of 16 for 10485760 nodes. The new memory size would allow a virtual processor ratio of 64 for 41943040 nodes.

Further research into data parallel techniques and their use in the solution of scattering problems is ongoing. These include an extension to 3-dimensional finite elements, effectiveness of the absorbing boundary condition for both 2- and 3-dimensional problems and convergence properties of the conjugate gradient algorithm when applied to complex geometries.

Also, with respect to mesh generation, several areas require further investigation. These include parallel mesh generation, mesh refinement techniques as well as other interpolation schemes. Mesh refinement techniques allow the program to actively alter the node distribution in the physical domain. This permits more nodes to be allocated in regions where the solution is expected to vary rapidly and fewer nodes in regions where the solution is expected to be relatively constant. Thus, a better approximation to the exact solution is obtained for a given number of nodes.

6 Acknowledgements

This work was supported by NSF grant # EET-8812958.

Computational resources were provided by Los Alamos National Laboratories, Los Alamos, New Mexico and by the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to, DARPA, and the Air Force Systems command, Rome Air Development Center (RADC), Griffiss AFB, NY, under contract number F30602-88-C-0031.

References

- [1] R.F. Harrington, Time Harmonic Electromagnetic Fields, McGraw-Hill: San Francisco, 1961.
- [2] A. Bayliss and E. Turkel, "Radiation boundary conditions for wave-like equations," Communications on Pure and Applied Mathematics, 33,707-725.
- [3] Thinking Machines Corp. Connection Machine Model CM-2 Technical Summary, Version 5.1, Thinking Machines Corp., 1989.
- [4] S.A. Hutchinson, S.P. Castillo and E. Hensel, "A basic finite element code on the Connection Machine," Proceedings of the Fourth Annual Conference on Hypercubes, Concurrent Computers and Applications, to be published, Monterey CA, 1989.
- [5] S.A. Hutchinson, S.P. Castillo and E. Hensel, "Solving 2-d electrostatic problems on the Connection Machine using the finite element method," Proceedings of the 5th Annual Review of Progress in Applied Computational Electromagnetics, Monterey CA, 1989.

- [6] R.E. Cline et al., "Towards the development of engineering production codes for the Connection Machine," Proceedings of the Fourth Annual Conference on Hypercubes, Concurrent Computers and Applications, to be published, Monterey CA, 1989.
- [7] S.L. Johnsson and K. Mathur, Data Structures and Algorithms for the Finite Element Method on a Data Parallel Supercomputer, Technical Report CS89-1, Thinking Machines Corp., 1988.
- [8] J.F. Thompson, Z.U.A. Warsi and C.W. Mastin, Numerical Grid Generation - Foundations and Applications, North Holland: New York, 1985.
- [9] M.R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solution of linear systems," J. Res. Nat. But. Standards, vol. 49, pp.409-436, 1952.
- [10] Thinking Machines Corp. Introduction to Programming in C/Paris, Version 5, Thinking Machines Corp., 1989.

Parallel Finite Elements Applied to the Electromagnetic Scattering Problem

R.D.Ferraro T.Cwik N.Jacobi P.C.Liewer T.G.Lockhart G.A.Lyzenga J.Parker J.E.Patterson

Jet Propulsion Laboratory/California Institute of Technology

Abstract

An 2D electromagnetic finite element analysis code which runs on the JPL/Caltech Mark IIIfp Hypercube is being upgraded to handle fully 3 dimensional scattering problems. The EM code uses finite elements [1] to model a finite problem domain which may include regions of anisotropic or nonuniform dielectric properties. It solves the single frequency source driven vector wave equation for electric or magnetic fields in this domain. The code is being implemented as a testbed for finite elements as applied to EM problems, with several types of elements, radiation boundary condition strategies, and parallel solvers.

Introduction

The general electromagnetic scattering problem is computationally taxing for even the most powerful present day computers. As depicted in Figure 1, the problem may be represented as a set of scatterers contained within a computational domain. The scatterers may be electrically complicated objects, consisting of various dielectric materials and conductors. The objects are illuminated by a known incident field, and the scattered electromagnetic radiation is to be determined. The accurate geometric representation of each object is necessary in order to correctly compute the field solution for the interesting case, i.e., when the wavelength of the incident radiation is comparable in size to the features of the scatterers themselves. There are several methods capable of solving the problem, each having different requirements for storage, computation, and geometric complexity. We have implemented a parallel EM analysis code which uses the finite element technique [1] to model the problem domain. Our choice of finite elements is based upon the need to accurately model the scatterers, while living with constraints of storage and cpu performance. A mildly complicated 3D problem can easily require the solution of 10⁶ simultaneous equations.

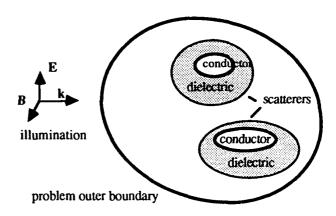


Fig. 1. The Canonical Electromagnetic Scattering Problem. An electric or magnetic field solution is required for the entire problem domain, which may include several scatterers consisting of many materials.

The finite element method decomposes the problem domain into a set of contiguous non-overlapping elements of various shapes which can support a piecewise continuous function with various degrees of smoothness. Linear finite elements can support piecewise linear functions, while higher order elements can support piecewise quadratic or cubic functions. The elements themselves can be shaped to conform to the problem geometry, and consist of a set of nodes at which are defined local orthonormal basis functions. These basis functions are zero at every other node, and have non-zero value only within the elements which contain the given

node. The problem solution is obtained by solving a set of linear equations for the coefficients of these basis functions, which usually represent the values of the solution at the nodal points. This is exactly like a fourier decomposition of some function, except that the orthogonal basis functions are local finite elements instead of sines and cosines. This has an important advantage for parallel processing - the basis functions interact only with their nearest neighbors and only require local knowledge of the problem domain. This is in sharp contrast to something like a fourier decomposition, where each basis function must sample the entire problem domain. The matrix equation which results from a finite element model is also extremely sparse, which translates into the ability to meticulously model the scatterer geometry with an appropriate density of elements, while using a more coarse mesh in less critical regions.

Code Structure

Parallelization of a finite element code is best achieved by partitioning the problem domain among the processors. The local nature of the finite element means that the matrix of linear equations can be assembled entirely in parallel, with each processor constructing the part of the matrix which corresponds to its subset of elements. The sparse matrix which results represents couplings among nearest neighbors on the finite element grid. These interactions can be visualized by looking at a sample finite element grid. In Fig. 2, a sample 4 node quadrilateral finite element grid has been partitioned (not optimally) among four processors. Each equation couples 1 node to its nearest neighbors through finite elements which share that node, so in the grid depicted, a node couples to at most 9 other nodes. For nodes in the interior of the subdomains (shown in black), the equation which represents it involves entries which are local to its parent processor. Those nodes on the subdomain surfaces require information which must be obtained from 2 or more processors. Thus a partitioning strategy for parallel processing must seek to minimize the surface area of these subdomains while dividing the problem domain into pieces of approximately equal volume.

To satisfy these requirements, we use a Recursive Inertial Partitioning Algorithm. The algorithm is designed for hypercube topology, where the number of processors is a power of 2. It proceeds as follows. First determine the "center of mass" of the grid. Then determine the axis through the center of mass with minimum moment of inertia. Bisect the grid with a plane through the center of mass which is perpendicular to this axis. Then repeat the procedure on each subgrid until the number of pieces

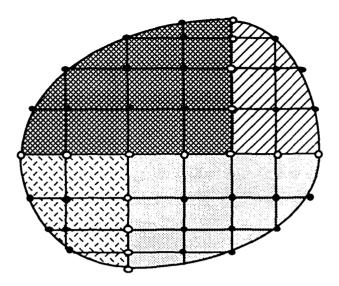


Fig. 2. A sample finite element domain decomposition. Each processor has a mutually exclusive subset of the elements. Nodes internal to the subdomains are exclusive to the parent processor, while nodes on the surfaces must be shared.

corresponds to the number of processors. Refinements to this decomposition can be made using simulated annealing to further minimize the surface areas of the subgrids, but the technique works best with user interaction. In most cases, the load balance and communication requirements which result by just doing the recursive inertial partitioning are sufficient.

The EM analysis code has been designed as a testbed for the finite element technique. To that end, we have implemented several types of finite elements, radiation boundary conditions, and parallel solvers. The number and sparsity of the linear equations set immediately suggest an iterative solver such as the Bi-Conjugate Gradient method [2]. This method requires only the computation of matrix-vector products and dot products, and with exact arithmetic will compute the solution in niterations, where n is the rank of the matrix. In practice, a sufficiently accurate solution can often be obtained in some fraction of n iterations. In our parallel implementation of the Bi-conjugate gradient algorithm, the matrix is never completely assembled. Each processor computes and retains matrix entries for only those nodes in its subdomain. Matrix entries corresponding to subdomain surface nodes are only partially assembled. Each processor does a piece of the matrix-vector multiply or dot product at each iteration step, with results being globally assembled only for the surface nodes. Since the computation required for a matrix-vector product or a dot product scales like the volume of the subdomains, while

the communication scales like the surface area, this algorithm becomes more efficient as problem size increases.

We are also investigating a hybrid bi-conjugate gradient / gaussian elimination algorithm. This method uses gaussian elimination to remove the interior nodes in favor of the surface nodes. Bi-conjugate gradients are then used to complete the solution. The gaussian elimination step can be done entirely without communication, and has the advantage of producing a partially inverted matrix. It has the disadvantage of requiring more storage, since sparsity is lost during the gaussian elimination step.

Results

The Electromagnetic Finite Element Code (EMFEC) consists of four major sections. During the input / initialization phase, the finite element model, hypercube partitioning information, and excitation parameters are read. The model is translated into a local node and element numbering scheme, and the communication routing is determined for the solver. In the current implementation, every processor must read the entire finite element model and partitioning information to determine which nodes and elements it requires. Element setup is done next. This consists of computing entries in the sparse stiffness matrix and entries in the force vector for all elements The solver is the third phase of computation. For the bi-conjugate gradient solver, iterations on an initial guess are performed until a solution with residual error below a specified tolerance is obtained. Finally, the solution is collected from all of the processors and written to a file.

Performance of each of these code sections, as well as the entire code itself, is illustrated in Fig.3. Here we plot speedup for a fixed test problem, consisting of a 2D finite element model of a dielectric cylinder with electric permativity $\varepsilon = 2.56$, and unit radius. The model contains 2304 quadrilateral elements, and a total of 9313 nodal points. We define speedup s as

$$s = T_s/T_p$$

where T_S is the execution time for the problem on one processor and T_p is the execution time on multiple processors. The element setup and solver sections of the code show almost uniform speedup as the number of processors is increased, achieving almost 75% efficiency on 32 processors. The output code section exhibits performance saturation almost immediately, due to the hardware bottleneck in transmitting data to the hypercube

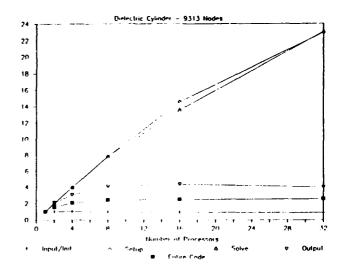


Figure 3. Speedup versus number of processors for a dielectric cylinder test case. The floating point intensive code sections exhibit uniform speedup with increasing processor numbers. The I/O related code sections show practically no speedup.

host. The input section is completely flat, since each processor must read the entire input data set. Poor performance of these two code sections have a devastating impact on the overall code performance. The entire code has a measured speedup of only 4 on 32 processors.

This effect is illustrated in Fig. 4, where we have plotted the percentage of execution time spent in each major code section as a function of the number of processors in use. Running on 1 processor, 65% of the execution time is spent in the compute phases of the code (element setup and solver) while the remaining 35% of execution time is spent in reading the model and writing the solution. For 32 processors, only 10% of the time is now spent on computation. The code has become I/O bound. Clearly any performance improvement must come from improving the I/O code sections, since they currently derive no benefit from the parallel architecture. Part of the problem is algorithmic in nature. The input phase of the code requires that each processor look at the entire finite element model. The poor performance of the output phase is either operating system or hardware related, however, since the code does not impose any ordering of the data being written to the output file.

We have also begun work on a 3D version of the code. Preliminary test results indicate that performance for each code section is unchanged for the larger models associated with 3D finite element problems. Overall code efficiency

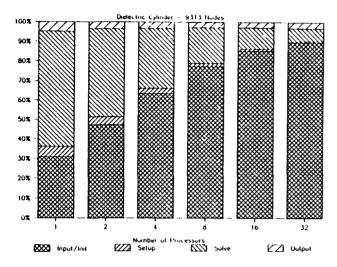


Figure 4. Percentage of execution time spent in each code section versus number of processors. Executing on 1 processor, 65% of the execution time is spend in computation. On 32 processors, the code is completely I/O bound.

is slightly improved, since the computation time for the solver scalars as n², where n is the number of nodes in the model, while the I/O time scales linearly with n. The larger finite element models resulting from 3D objects account for the overall improvement. None the less, a better system for reading the model is called for. We are investigating the possibility of assigning node and element data to processors in a card dealing fashion, then rearranging the data by using recursive inertial partitioning in parallel. The absolute bottleneck of transmitting data to or from the host computer cannot be avoided. However, by not requiring every processor to examine the entire model, we expect that we can obtain some improvement in the efficiency of the input phase of the code.

Conclusions

We have demonstrated that the computational phase of a finite element code can be performed efficiently on a concurrent computer like the Mark IIIfp Hypercube. The I/O bottleneck in transferring the large datasets to and from the host computer remains a problem, limiting the overall efficiency of the code. We are working on ways of mitigating the effect of this bottleneck, though hardware constraints will ultimately prevent us from eliminating it.

References

- [1] T.J.R. Hughes, <u>The Finite Element Method</u>, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987
- [2] D.A.H. Jacobs, "The Exploitation of Sparsity of Iterative Methods", in Sparse Matrices and Their Uses, edit. by I.S. Duff, Academic Press, London, (1981)

An Examination of Finite Element Formulations and Parameters for Accurate Parallel Solution of Electromagnetic Scattering Problems

J. W. Parker, R. D. Ferraro, P. C. Liewer

Jet Propulsion Laboratory/California Institute of Technology

Abstract

In conjunction with the development of a test bed for finite element approaches for solving three-dimensional electromagnetic scattering problems on the JPL/Caltech Mark IIIfp Hypercube, we have developed guidelines for choosing parameters which improve accuracy at the cost of computational resources. We show results of a series of numerical experiments through which we can now predict the required mesh density and the radius of a given type of absorbing boundary which are needed to obtain virtually any desired degree of accuracy for a given finite element type. This type of guideline set is important for threedimensional finite element computations, because the number of unknowns scales as the cube of both the linear node density per wavelength, and the radius of a spherical absorbing boundary. We examine several finite element formulations, including both node- and edgebased elements, and discuss their relative merits for accuracy vs. computational resources, ease of parallel implementation, and parallel efficiency.

Introduction

The finite element method is well suited to electromagnetic scattering problems in which the scattering object is not too large, but of the most general linear sort. The method can accurately model EM fields in domains containing conductors, lossy and lossless dielectric and magnetic materials, anisotropic materi-

als, and extremely inhomogeneous materials. The method is also well suited to parallel computation: with elements spatially partitioned among the processors, the filling of a distributed stiffness matrix, and its iterative solution require very little interprocessor communication. The expanding capacity of parallel computers opens the vista of accurate ever larger problems. solution to However, the achievement of accurate solutions often depends on the adequate choice of more than one resource-intensive parameter. Open-region scattering requires truncation of the domain with some sort of absorbing boundary condition. The solution gains accuracy as the truncation is placed farther from the object, requiring the solution to a larger system of equations. This added cost will buy nothing if the accuracy is near the limits implied by other Assuming sufficient care is taken in modeling the object and performing numerical integrations, the remaining factors of interest are the formulation of the finite element, and the element spatial density. The latter is the most resource-intensive. Ideally, one uses the most accurate element formulation for the problem, then chooses the element density and the size of the truncated domain such that the accuracy limitations of each are balanced.

Finite Element Formulations

Three types of electromagnetic scattering problems are addressed by analysis codes which run on the JPL/Caltech Mark IIIfp Hypercube. The simplest code

solves the 2-D scalar Helmholtz equation, using finite elements filling a circular truncated domain. The numerical technique is that used by the sequential code of [1]. The absorbing boundary condition used is the second-order condition of [2]. Several finite element formulations are including supported, linear and quadratic elements, triangular and elements, and total field quadrilateral and scattered field elements. Another code solves the 2-D vector Helmholtz equation, and is primarily a test-bed for investigating various element types. The vector problem admits several additional types of element formulation, including analogues of the scalar problem finite element varieties, but also element types implying different means of representing the vector components. Node-based vector elements solve for the orthogonal vector components as two independent unknowns. Edge-based elements solve for the tangential component of the field along each element edge. To date, the code supports node and edge elements. Even more exotic element types are possible. The final code solves the full 3-D vector Helmholtz equation, using analogues of the 2-D vector elements.

Parametric Study

We have performed a parametric study of the solution accuracy for a canonical problem using the 2-D scalar code. The problem studied is the scattering from a dielectric 2.56 cylinder with radius a and wave number k such that ka = 1. This problem has a well-known analytic solution, which we used to measure the accuracy of the finite element solution. Through trial and error, we found a combination of element formulation. element density, and truncated domain size which resulted in extremely high We then systematically reaccuracy. duced the element density and the domain size independently, to determine the accuracy impact of each factor separately. The element type used is the quadratic, isoparametric (i. e., curved) 9 node (Lagrangian) quadrilaterals

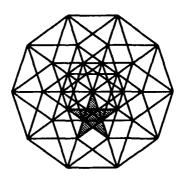
modelling the scattered field. This element produced more accurate results than the quadratic triangle element and the total field elements, and was a considerable improvement over any linear element. The element density is 15.7 per wavelength, and the domain radius r is such that kr = 4.2. The accuracy is such that one relative error measure (the magnitude of the far-field error at each angle divided by the maximum magnitude of the far field) is $< 10^{-4}$. This implies an error in the RCS of < 1/10 dB over a 60 dB range. The parametric study results in curves of the relative error vs. kr and kh, where h is the minimum finite element side. Note that in this case k must be interpreted as the wavelength inside the material; for example, high dielectric materials require a finer grid, by a factor of the square root of the relative dielectric. Both curves display power-law characteristics. The kh curve goes as $(kh)^3$, which matches the theoretical behavior of field RMS error for quadratic elements. The kr curve goes approximately as $(kr)^{-3}$. but we know of no justification of this behavior from numerical theory. We believe the kh behavior represents a reliable rule for any size and composition of scattering object, while the krbehavior may apply only to this problem. To obtain a rough rule of thumb for size of the truncation domain, a large scattering object was modeled: a perfect conducting circular cylinder ka = 50. Good agreement (within 2 dB) with the analytic RCS was obtained by using kr = 62, while kr = 56 was not considered adequately accurate (errors exceed 4 dB). At the level of accuracy represented by the kr = 62 case, using more than 3 or 4 quadratic elements per wavelength proves wasteful. We conjecture that with the use of the secondorder Bayliss Turkel condition, using r about 25% larger than the object halfdiameter will generally produce RCS curves with similarly high quality.

Vector Elements

Similar accuracy studies of vector elements are proceeding, and results will be reported soon. A comparison has been made between linear triangular edge elements and node-based elements for the ka = 1, dielectric 2.56 circular cylinder, using 30 elements per wavelength and a modified Sommerfeld boundary condition at kr = 3. The node-based elements produced a more accurate RCS. Both types of element are convenient for parallel partitioning, with the level of inconvenience depending on the form of model specification. For mesh generators which produce a model in the form of an ordered list of nodal coordinates and element nodes, the edge elements require an extra computational step: compiling a list of numbered edges owned by each element. With respect to a given triangular mesh, the edge elements should imply a slightly faster solution: a mesh node is shared by six elements, while an edge is shared by only two, implying both a sparser matrix and less communication for the edge elements. However, this advantage may prove to be outweighed by considerations of accuracy.

References

- [1] Peterson, A. F. and S. P. Castillo (1989), A Frequency-Domain Differential Equation Formulation for Electromagnetic Scattering From Inhomogeneous Cylinders, *IEEE Trans. Antennas Prop.*, 37, 601-607.
- [2] Bayliss, A. and E. Turkel (1980), "Radiation boundary conditions for wave-like equations," Comm. Pure Appl. Math., 33, 707-725.



The Fifth Distributed Memory Computing Conference

17: Plasma Physics Applications

Massively Parallel Fokker-Planck Calculations*

Arthur A. Mirin

National Magnetic Fusion Energy Computer Center Lawrence Livermore National Laboratory Livermore, California 94550

Abstract

The Fokker-Planck package FPPAC [1,2], which solves the complete nonlinear multispecies Fokker-Planck collision operator for a plasma in twodimensional velocity space, has been rewritten for the Connection Machine 2. This has involved allocation of variables either to the front end or the CM2. minimization of data flow, and replacement of Crayoptimized algorithms with ones suitable for a massively parallel architecture. Coding has been done utilizing Connection Machine Fortran. Calculations have been carried out on various Connection Machines throughout the country. Results and timings on these machines have been compared to each other and to those on the static memory Cray-2 at the National Magnetic Fusion Energy Computer Center. For large problem size, the Connection Machine 2 is found to be cost-efficient.

Introduction

Over the past several decades there has been a tremendous increase in computer power. Most of this increase has been due to improvements in hardware. Electrical components have now become so efficient, however, that more recently the concentration has been on improving the architecture of the computer. This had led to shared memory multiprocessor supercomputers such as the Cray-2 and the Cray-YMP.

Although these multiprocessor devices have had a substantial impact on high speed computing, it has been recognized that a more cost-effective approach might be to link together very large numbers of slower, cheaper processors, each with its own local memory. Such massively parallel computers, although not at this time general purpose, have performed quite impressively in a number of problem areas and are being used selectively in production mode. Many members of the computing

community are beginning to think in terms of their use as general production machines in the not-too-distant future.

There are fundamentally two different types of massively parallel architectures—single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD). In an SIMD device all of the processors execute the same instruction in lock-step fashion, whereas in an MIMD device each processor may follow its own set of instructions. Programming an SIMD machine is analogous to vectorizing (on a Cray), whereas programming an MIMD machine amounts to true multitasking. Quite naturally, SIMD machines are cheaper per processor and inherently easier to program. However, they generally demand greater parallelism in the algorithm and offer less flexibility. The most prominent example of an SIMD device is the Connection Machine 2 (CM2), the computer used in this investigation. Manufacturers of MIMD machines include INTEL, NCUBE, and BBN.

This study involves the massive parallelization of a code known as FPPAC [1,2], which time-integrates the Fokker-Planck collision operator in a plasma. Codes such as FPPAC are used to simulate collisional phenomena in magnetically confined plasmas, particularly in situations where the charged particle distribution functions depart sufficiently from Maxwellians. Such scenarios include the heating of a plasma by radiofrequency waves or energetic neutral beams, and the loss of particles from selected areas of velocity space [3]. The relevant equation is the Boltzmann equation with Fokker-Planck collision terms [4], more commonly referred to as the Fokker-Planck equation.

The problem is to solve a nonlinear partial differential equation for the distribution function of each charged species in the plasma in terms of six phase space variables (plus time). However, certain symmetry and ordering assumptions can often be made, allowing the

^{*}Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

dimensionality to be reduced to four—two spatial and two velocity coordinates. When the magnetic field is uniform or when the particle bounce motion operates on a time scale sufficiently faster than other phenomena of interest, one spatial variable may be eliminated, reducing the dimensionality to three. And when diffusion across flux surfaces is slow relative to velocity space dynamics, the other spatial coordinate may also be eliminated, leaving only two velocity coordinates.

FPPAC invokes the above assumptions and solves the complete nonlinear multispecies Fokker-Planck collision operator for a plasma in two-dimensional velocity space. The operator is expressed in terms of spherical coordinates [speed (v) and pitch angle (θ)] under the assumption of azimuthal symmetry. Provision is made for additional physics contributions (e.g., sources and losses, radio-frequency heating, electric field acceleration). The charged species, referred to as general species, are assumed to be in the presence of an arbitrary number of fixed Maxwellian species. The electrons may be treated either as one of those Maxwellian species or as a general species. Coulomb interactions among all charged species are considered.

The Fokker-Planck Collision Operator

The Fokker-Planck collision operator may be expressed in the form

$$\left(\frac{\partial f_{a}}{\partial t}\right)_{c} = \Gamma_{a} \left\{ -\frac{\partial}{\partial v_{i}} \left(f_{a} \frac{\partial h_{a}}{\partial v_{i}} \right) + \frac{1}{2} \frac{\partial^{2}}{\partial v_{i} \partial v_{i}} \left(f_{a} \frac{\partial^{2} g_{a}}{\partial v_{i} \partial v_{i}} \right) \right\} \tag{1}$$

where f_a is the distribution function of species a and Γ_a is a constant [1]. The Rosenbluth potentials [4] g_a and h_a are written as

$$g_{a} = \sum_{b} \left(\frac{Z_{b}}{Z_{a}}\right)^{2} \ln \Lambda_{ab} \int f_{b}(v') |v - v| dv' (2)$$

$$h_{a} = \sum_{b} \left(\frac{m_{a} + m_{b}}{m_{b}}\right) \left(\frac{Z_{b}}{Z_{a}}\right)^{2}$$

$$\ln \Lambda_{ab} \int f_{b}(v) |v - v|^{-1} dv' \qquad (3)$$

Under the assumptions stated above, Eq. (1) may be written in the form

$$\frac{1}{\Gamma_{\mathbf{a}}} \left(\frac{\partial f_{\mathbf{a}}}{\partial t} \right)_{\mathbf{c}} = \frac{1}{\mathbf{v}^2} \frac{\partial}{\partial \mathbf{v}} \left(A_{\mathbf{a}} f_{\mathbf{a}} + B_{\mathbf{a}} \frac{\partial f_{\mathbf{a}}}{\partial \mathbf{v}} + C_{\mathbf{a}} \frac{\partial f_{\mathbf{a}}}{\partial \theta} \right) \\
+ \frac{1}{\mathbf{v}^2 \sin \theta} \frac{\partial}{\partial \theta} \left(D_{\mathbf{a}} f_{\mathbf{a}} + E_{\mathbf{a}} \frac{\partial f_{\mathbf{a}}}{\partial \mathbf{v}} + F_{\mathbf{a}} \frac{\partial f_{\mathbf{a}}}{\partial \theta} \right) \tag{4}$$

where the coefficients A_a , B_a , C_a , D_a , E_a , and F_a are expressible as linear combinations of the Rosenbluth potentials g_a and h_a and their various derivatives. The quantities f_a , g_a , and h_a are then expanded in Legendre polynomials $P_l(\cos \theta)$, with the result that the coefficients of the series for g_a and h_a may be written in terms of moments of the coefficients of the series for the various distribution functions. When the dust clears, the six coefficients of Eq. (4) are expressed as linear combinations of functionals of the type $M_l(w)$, $N_l(w)$, $R_l(w)$, and $E_l(w)$, where w is a coefficient of a Legendre series for a distribution function; those four functionals are given in Eqs. (5)–(8) below:

$$M_l(w)(v) = \int_{v}^{\infty} w(y) y^{(1-l)} dy$$
 (5)

$$N_l(w)(v) = \int_0^v w(y) y^{(2+l)} dy$$
 (6)

$$R_l(w)(v) = \int_{v}^{\infty} w(y) y^{(3-l)} dy$$
 (7)

$$E_l(w)(v) = \int_0^v w(y) y^{(4+l)} dy$$
 (8)

Further details are given in McCoy, et al. [1].

Spatial Representation

A variably spaced finite-difference grid in ν and θ is employed. Numerical differentiation is carried out using nearest neighbors. The boundary conditions are expressed in an analogous manner. The resulting scheme conserves particle density down to round-off.

Temporal Discretization

Equation (4) is time-integrated using either implicit operator splitting, an alternating direction implicit (ADI) method, or fully implicit differencing. The former two are appropriate for time-dependent simulations, whereas the fully implicit method (which has not yet been implemented for the Connection Machine) is most optimal for approaching steady state.

The Connection Machine 2

The CM2 consists of up to 65536 single bit processors, each with 8 to 32 kbytes of random access memory. The processors are stored 16 to a chip, each pair of chips sharing Weitek Floating point hardware. Altogether there are up to 4096 chips arranged in a 12dimensional hypercube. Arithmetic is typically performed on 32-bit data, although using 64 bits (double precision) is allowed. Chips which have actual 64-bit arithmetic have recently become available. An important feature of the CM2 is its support for virtual processors; that is, if one wishes to execute 64K operations in parallel on a 16K machine, he may assign 4 virtual processors (each with one quarter the memory) to an actual processor. The most general form of communication among the processors is the router. Nearest neighbor communications, however, can be handled much more efficiently using a separate communications mechanism called the NEWS grid.

The CM2 is not a stand-alone machine. It is typically front-ended by a VAX, a SUN-4, or a Symbolics. The front end, which is a serial machine, stores scalars and short arrays and provides instruction sequencing and some I/O. One may program in either of three higher-level languages—CM Fortran, C*, or *LISP; these are extensions of Fortran, C, and Common LISP, respectively. Alternatively, one can use the PARallel Instruction Set (PARIS) for CM operations together with either of the above three languages on the front end. (Or for that matter, PARIS instructions may be embedded in the higher level CM languages.) Of the various CM languages, CM Fortran is the newest, and it requires the VAX front end.

CM Fortran

CM Fortran is one of the first Fortran implementations using 8X constructs. Of particular importance are the array constructs, which are designed to generate parallel code. For example, the Fortran 8X code block in Table 1 accomplishes the same task as the Fortran 77 beneath it.

Table 1. Comparison of Fortran 8X and Fortran 77

Fortran 8X

WHERE (D .NE. 0.)
A=EOSHIFT(B,1,1)+.5*EOSHIFT(C,2,-1)
ELSEWHERE
A=D
ENDWHERE

Fortran 77

DO 1 I=1,100 DO 1 J=1,100 IF (D(I,J) .NE. 0) THEN A(I,J)=B(I+1,J)+.5*C(I,J-1) ELSE A(I,J)=D(I,J) ENDIF 1 CONTINUE

On the Connection Machine, by default, corresponding elements of arrays of the same shape are stored on the same processor. The interprocessor communications in the example above are taken care of by the EOSHIFT intrinsic (which in this case uses high speed nearest neighbor communications), and the WHERE construct (which is analogous to CVMGT in Cray Fortran) allows parallelization of the loop. The more compact style makes coding easier to read and debug.

Conversion of FPPAC to the Connection Machine

The conversion of FPPAC has involved a number of steps, including the following:

- (a) Allocating variables either to the front end or the CM. As a rule, large arrays are stored on the CM and everything else is on the front end. To save communications costs, a number of 1-D arrays are SPREAD into two dimensions [e.g.,XZ(I,J)=X(J)].
- (b) Minimizing data flow between the front end and the CM. Special routines can be used to transfer blocks of data.
- (c) Converting to 8X constructs. Treating the Cray as home base for FPPAC, the use of .IF statements (which are processed by the CTSS precompiler) enables the mimicking of 8X constructs with

- 77 counterparts side by side. The Cray version can then be debugged prior to testing on the Connection Machine.
- (d) Replacement of Cray-optimized algorithms with highly parallel ones. The two principal areas involved are the Fokker-Planck coefficient computation and the time integration procedure.

Computation of Fokker-Planck Coefficients

The computation of the Fokker-Planck coefficients involves the following steps:

- (a) Computation of Legendre projections of the distribution functions.
- (b) Computation of the moments [Eqs. (5)–(8)].
- (c) Computation of Legendre projections of the coefficient pieces.
- (d) Summation of the Legendre series.

Steps (a) and (d) (which turn out to be the most timeconsuming) are cast in terms of matrix multiplication. Step (c) involves linear combinations of the various moments. A typical term in step (b) involves computing a set of quantities of the form

$$\alpha_{j} = \int_{0}^{\nu_{j}} f(\nu) \ d\nu \tag{9}$$

for j=1,...J. Using the Cray methodology, this takes on the order of J sequential (though partially vectorizable) operations. On the CM, however, this procedure is recast into one which takes on the order of log J parallel steps, thus saving alot of work. One computes the numbers

$$\beta_j = \int_{\mathbf{v}_{j-1}}^{\mathbf{v}_j} f(\mathbf{v}) d\mathbf{v}$$
 (10)

and then combines them in an appropriate manner. This type of procedure is called a "SCAN." The total number of arithmetic operations is greater, but due to the inherent parallelism of the hardware, the number of actual (parallel) steps is smaller. Furthermore, the communications steps, which involve meshpoint indices differing by a power of two, are carried out using either a small number of nearest neighbor jumps or up to two "hops" along the router (by virtue of the binary reflected gray code ordering of NEWS arrays).

Time-Integration Procedure

Either of the two schemes (implicit operator splitting or ADI) involve the solution of parallel tridiagonal systems. On the Cray, vectorization over the direction orthogonal to the sweep is carried out. On the CM, a procedure known as parallel cyclic reduction [5] is also implemented. During each step of the ordinary cyclic reduction procedure, both odd and even reductions are carried out (in parallel), so that no backfilling is necessary once the reduction procedure is complete. As with the moments computation, this takes a greater number of arithmetic operations but a smaller number of parallel steps.

Connection Machine Facilities

The debugging of the Connection Machine version of FPPAC was carried out primarily at the Advanced Computing Research Facility (ACRF) at Argonne National Laboratory. Pre-released features of the Fortran compiler were tested on the CMNS computer at Thinking Machines Corporation (TMC). Timing comparisons were carried out on the Connection Machines at the National Aeronautics and Space Administration Ames Research Center (NASA Ames) and the Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory, primarily the former. Timings presented in this work are for the NASA Ames facility, unless otherwise stated. An arrangement has just been made to use the Connection Machine at Florida State University (FSU). A comparison of the various Connection Machines is given in Table 2.

Table 2. Comparison of Connection Machine Facilities

Facility	No. of Processors	Hardware Upgrade	Vax Front End	
ACRF	16K	no	8650	
TMC	32K	no	6250	
NASA	32K	no	6320	
ACL	64K	in progress	6420	
FSU	64K	yes	6420	

Cray Facility

The Connection Machine calculations are compared to ones utilizing the Cray-2 "F" machine at the National Magnetic Fusion Energy Computer Center (NMFECC) at Lawrence Livermore National Laboratory (LLNL). This Cray has a static memory of 128K megawords and like all others at NMFECC, runs the Cray Time Sharing System (CTSS). The Cray Research Incorporated CFT77 compiler is invoked.

Coarse Mesh Calculation

The first case considered has a velocity-space mesh consisting of 128 points in ν and 64 points in θ . The Fokker-Planck coefficients are expanded in a 5-term Legendre series. The calculation is run using 8K processors and 32-bit arithmetic. (Note that 64-bit arithmetic is mandatory on the Cray-2.) A timing comparison is shown in Table 3. All times are given in minutes. The headings "CM-total" and "CM-active" refer to the total elapsed time and the time the Connection Machine (CM) is active, respectively.

It can be seen that the time advancement takes 8 times as long on the CM as on the Cray. Assuming a full Cray to cost four times as much as a full CM, this translates to comparable cost efficiency. Note that the virtual processor (VP) ratio for the CM case is unity. It may also be observed that the coefficient computation takes 200 times as long on the CM as on the Cray. This is due to the fact that the degree of parallelism of the calculation is at most 640 and to the fact that the CM matrix multiply routine (MATMUL) operates quite inefficiently on small matrices. Clearly, the Connection Machine is not suitable for a case having only 5 Legendre polynomials.

Longer Legendre Series

The number of Legendre polynomials is now increased from 5 to 64. This allows the coefficients calculation to have a degree of parallelism equal to 8K, except per-

Table 3. Timing Comparison for Coarse Mesh Case

Procedure	Cray-2	CM- total	CM- active
Coefficients Time advancement		1.3×10^{-1} 2.5×10^{-3}	

haps for the matrix multiply, in which typically a 128 by 64 matrix multiplies a 64 by 64 matrix. The increase to 64 Legendre polynomials requires parts of the calculation to be carried out in double precision. More specifically, the computation of the moments requires powers of v; the higher the Legendre polynomial, the higher the power of v. Of critical importance is the allowable exponent range available to represent the powers of v, rather than the accuracy. A time comparison is shown in Table 4.

It can be seen that the coefficients computation now takes only 48 times as long on the Connection Machine as on the Cray. Virtually all of the time (over 99%) is spent in the matrix multiply. The double precision part of the calculation, namely the time it takes to compute the moments, is insignificant.

Aside: Exponent Range

It is important for users of the Connection Machine to be aware of an inconsistency in double precision representation. The CM double precision implementation follows IEEE standards and provides for an 11-bit representation of the exponent. The double precision representation on the Vax front end, however, provides only 8 bits for the exponent, thereby severely limiting the exponent range on the front end. One must exercise great care when transferring double precision data between the Connection Machine and the Vax and when doing seemingly "harmless" double precision operations. It is of further interest to note that Cray single precision allocates 15 bits for the exponent, enabling the Cray to represent a wider range of numbers but at less accuracy.

Matrix Multiplication

The speed of the CM Fortran matrix multiply (MATMUL) limits the performance of the Connection Machine in the above case with 64 Legendre polynomials. It is therefore of interest to compare matrix multiply efficiency as a function of matrix size. Such a

Table 4. Timing Comparison for Coarse Mesh Case with Long Legendre Series

Procedure	Cray-2	CM- total	CM- active
Coefficients Time advancement	6.7×10^{-3} 3.2×10^{-4}	3.2×10^{-1} 2.5×10^{-3}	2.3×10^{-1} 1.4×10^{-3}

comparison is shown for square matrices in Table 5. The arithmetic uses 32 bits, and 16K processors are utilized.

It can be seen that the CM2 matrix multiply performance is very strongly dependent on the size of the matrix. At order 1024, the CM (in single precision) outperforms the Cray. However, at order 128, a case in which the number of matrix elements equals the number of processors, the CM operates 6 times more slowly.

Fine Mesh Calculation

The mesh is now expanded to 512 points in ν and 256 in θ . The Legendre series for computing the Fokker-Planck coefficients has 64 terms (the upper limit for a 512-point ν -mesh is approximately 111). The Connection Machine calculation is carried out with 16K processors. Thus, the VP ratio of the time-advancement phase is 8 (instead of 1). A timing comparison is shown in Table 6.

It can be seen that the time-advancement phase now executes almost as fast on the Connection Machine as on the Cray. This is due primarily to the higher VP ratio, Assuming only 32 bits to be required, this translates to a factor of 3 cost-effectiveness in favor of the CM. Furthermore, the coefficients computation executes 6 times faster on the Connection Machine, due both to the higher VP ratio and to the superior performance of MATMUL for larger matrices. In fact the matrix multiplication phase, although it still dominates, now is responsible for only 85% of the coefficients calculation. It may also be observed that the Connection Machine

Table 5. Matrix Multiply Performance

Order	Cray-2 Mflops	CM2 Mflops		
64	431	10		
128	438	76		
256	446	180		
512	452	319		
1024	453	577		
2048	454	too large for 16K		

Table 6. Timing Comparison for Fine Mesh Case

Procedure	Cray-2	CM- total	CM- active
Coefficients Time advancement	1.8×10^{-1} 7.3×10^{-3}	3.0×10^{-2} 1.0×10^{-2}	$2.8 \times 10^{-2} \\ 1.0 \times 10^{-2}$

experiences relatively less idle time; this is due to the higher VP ratio.

There are instances where the precise change in density will strongly affect the ensuing physics. In such cases 64-bit arithmetic is required. A comparison of "standard" and double precision is given in Table 7. Recall that the "standard" calculation uses single precision almost everywhere, but utilizes double precision when computing the moments.

It can be seen that the time advancement in double precision takes about 9 times longer than the single precision version. That is because the Connection Machines

Table 7. Standard Calculation versus Full Double Precision for Fine Mesh Case. (The initial density and energy are $6.8 \times 10^{+13}$ and 49.2228, respectively.)

(a) Procedure	CM-total (Std.)	CM-active (Std.)	CM-total (D.P.)	CM-active (D.P.)	
Coefficients	3.0×10^{-2}	2.8×10^{-2}	5.8 × 10 ⁺⁰	$5.8 \times 10^{+0}$	
Time advancement	1.0×10^{-2}	1.0×10^{-2}	9.2×10^{-2}	9.2×10^{-2}	
(b) Precision		Final Density		Final Energy	
Standard		$7.65638 \times 10^{+13}$		45.9036	
Double Precision		$7.64600 \times 10^{+13}$		45.9139	
Cray		$7.64600 \times 10^{+13}$		45.9139	

used for these tests have available only a software double precision implementation. It is intended to run tests on a Connection Machine having 64-bit arithmetic as soon as one becomes accessible. The time to compute the coefficients increases by a factor of about 200, due to both the software implementation of double precision and the lack of a high speed double precision matrix multiply package.

It can also be seen that the double precision answers are in better agreement with those of the Cray, which is to be expected. Since this test case is run for only ten timesteps, very little inference can be drawn from the difference in accuracy.

Timing on the Connection Machine

Timing Fortran programs on the Connection Machine is more of an art than a science, at least in comparison to timing Cray code blocks. That is because the Vax real time clock has a very low resolution (milliseconds) and because it measures the time consumed by all processes, not just the one in question. Furthermore, the CM-active time is computed by subtracting the CM-idle time from the total elapsed time as measured on the front end. Hence, the active time is no more accurate than the total elapsed time.

TMC advises users, when performing timing tests, to (a) use a lightly loaded front end system, (b) time code blocks whose duration is 1 to 5 seconds, and (c) run the code segment at least 5 times and use the minimum value reported [6].

Because the CM is in essence a slave of the front end, its overall performance will vary with the front end model. Generally speaking, the systems at TMC, NASA Ames, and ACL give roughly comparable performance; the Connection Machine at ACRF is not as robust.

Toward the Future

The version of FPPAC running on the Connection Machine contains most of the important features of the Cray version. Yet to be implemented are capability to treat a multiple number of species and a fully implicit finite-difference solver. The multispecies capability was left out of this version because of compiler limitations having to due with processor allocation for multiply dimensioned arrays. These limitations have recently been removed, however, and generalization to

multiple species should be simple and straight-forward. Provision of a fully implicit solver requires a parallel routine to compute the nine-banded operator matrix and a routine to solve that matrix (e.g., preconditioned conjugate gradient). Such a procedure is likely to dominate the calculation, so that the overall code performance will strongly reflect that of the fully implicit solver.

FPPAC was chosen for conversion because it is simple and because it is representative of more involved plasma Fokker-Planck models. Present state-of-the-art Fokker-Planck calculations can treat two spatial dimensions (one of them averaged over the particle bounce motion) as well as two velocity dimensions and contain a whole host of other physics as well. Extensions of this work to more realistic scenarios is under investigation.

Acknowledgment

I thank the National Aeronautics and Space Administration Ames Research Center, Los Alamos National Laboratory, Argonne National Laboratory, and Thinking Machines Corporation for use of their Connection Machine facilities. I especially thank Kyra Lowther of Thinking Machines Corporation for her very adept and prompt assistance with the Connection Machine at NASA Ames.

References

- [1] McCoy, M.G., Mirin, A.A., & Killeen, J. (1981) "FPPAC: A Two-Dimensional Multispecies Nonlinear Fokker-Planck Package," *Comput. Phys. Commun.*, 24(1):37-61, September.
- [2] Mirin, A.A., McCoy, M.G., Tomaschke, G.P. & Killeen, J. (1988) "FPPAC88: A Two-Dimensional Multispecies Nonlinear Fokker-Planck Package," Comput. Phys. Commun., 51(3):373-380, November.
- [3] Killeen, J., Kerbel, G.D., McCoy, M.G. & Mirin, A.A. (1986) Computational Methods for Kinetic Models of Magnetically Confined Plasmas, Springer-Verlag, New York.
- [4] Rosenbluth, M.N., MacDonald, W.M. & Judd, D.L. (1957) "Fokker-Planck Equation for an Inverse-Square Force," Phys. Rev., 107(1):1-6, July.
- [5] Hockney, R.W. & Jesshope, C.R. (1988) Parallel Computers 2, Adam Hilger, Bristol.
- [6] Thinking Machines Corporation (1990) CM Fortran Release Notes, Version 5.2-0.7 Beta, Thinking Machines Corporation, Cambridge, MA.

Implementing Particle-In-Cell Plasma Simulation Code on the BBN TC2000

Judy E. Sturtevant *
Mission Research, Inc.
1720 Randolph Rd. SE
Albuquerque, NM 87106

Arthur B. Maccabe[†]
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131

Abstract

The BBN TC2000 is a multiple instruction, multiple data (MIMD) machine that combines a physically distributed memory with a logically shared memory programming environment using the unique Butterfly switch. Particle-In-Cell (PIC) plasma simulations model the interaction of charged particles with electric and magnetic fields. This presentation describes the implementation of both a 1-D electrostatic and a 2 1/2-D electromagnetic PIC (particle-in-cell) plasma simulation code on a BBN TC2000. Performance is compared to implementations of the same code on the shared memory Sequent Balance and distributed memory Intel iPSC hypercube.

Introduction

In recent years the traditional model of single-processor, sequential computer archtecture has become known as the von Neumann bottleneck [13]. A single CPU issuing sequential requests over a bus to memory, and the memory responding to one request at a time creates the bottleneck. In response to this problem, designers, seeking alternatives to the von Neumann architecture, have developed a wide variety of parallel architectures and interconnection technologies. The BBN TC2000 is a multiple instruction, multiple data (MIMD) machine that combines a physically distributed memory with a logically shared memory programming environment using the unique Butterfly switch. Processors are connected through the Butterfly switch network. Data may be local to a processor, or remote (i.e., fetched through the switching network from another processor).

This presentation includes a discussion of the imple-

mentation of both a 1-D and a 2 1/2-D PIC (particle-in-cell) plasma simulation code on a BBN TC2000 at Argonne National Laboratory's Advanced Computing Research Facility. Performance is compared to implementations of the same code on the shared memory Sequent Balance and distributed memory Intel iPSC hypercube.

Architecture Overview

The most commonly used classification scheme in parallel computing is that of Flynn, which is based on the concepts of instruction streams and data streams.

SISD single instruction, single data
SIMD single instruction, multiple data
MISD multiple instruction, single data
MIMD multiple instruction, multiple data

The category of MIMD architecture has been subdivided into the categories of distributed and shared memory [13]. In addition, both distributed and shared memory may be further categorized by implementation, actual physical (hardware) implementation or logical (software) implementation [2]. The BBN TC2000 combines hardware (the Butterfly switch) with software to implement a logically shared memory programming environment on top of physically distributed memory. The logically shared memory model provides ease of programming, while the physical distribution of memory permits expandability. Physically shared memory systems are limited by a single bus with fixed bandwidth interconnecting processors and memory. Splitting memory into physically disparate modules, and providing multiple interconnection paths between processors and memory modules, allows expandability, at least up to several hundred processors.

The nX operating system provides the cluster mechanism for designating a number of function boards as a computing resource. The system cluster includes all the function boards of a machine. A number of function boards are allocated to a public cluster, used for

^{*}email: judy@unmvax.cs.unm.edu. This work was supported in part by the USAF Weapons Laboratory contract F29601-87-C-0054. It was completed as a graduate student independent study project in the Computer Science Department at the University of New Mexico.

[†]email: maccabe@unmvax.cs.unm.edu

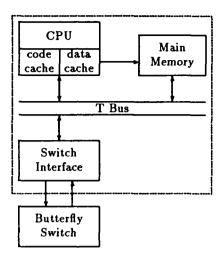


Figure 1: Function Board Components

compiling, editing, etc., and an I/O cluster containing the nX master function board. The cluster concept provides a flexible, multi-user environment.

The main components of the TC2000 architecture are function boards (8 to 504), connected by a Butterfly switch. A function board must include a switch interface, T-bus and TCS (Test and Control System) slave. To this minimum configuration may be added a processor, 4-32 Mbytes of memory and/or a VMEbus interface [3]. Processor boards are based on the Motorola 88000, which consists of an 88100 RISC CPU and two 88200 cache/memory management units, one for data and another for instructions. Connecting the components on the function board is the transaction bus, or T-bus, a 32 bit-wide memory bus with 80 Mbytes/sec peak bandwidth. Figure 1 illustrates the major components of a function board.

Each processor executes an independent sequence of instructions, referencing data as needed. Virtual addresses are translated by the memory management unit into physical addresses, which are in turn translated by the CPU interface into System Physical addresses. The interprocessor network allows each processor to share some or all of the system memory. Memory that is physically local to Processor #1 is considered remote by Processor #2, and vice versa. Code, constants, and stack variables are stored in local memory, not fetched across the network. The application data is usually spread across the memory of the machine by the Uniform System, but may be placed more explicitly by the programmer.

The unique component of the TC2000 architecture is the Butterfly switch. The Butterfly switch implements a packet switching network of 8-bit wide switch paths, with a bandwidth of 38 Mbytes/sec. Each processor is connected to the switch through an interface with two ports. One port is used to access other function boards, and the other is used to service requests from other boards. A remote memory access is one made over the switch; a local memory access is one that accesses its own function board directly. Multiple memory accesses in parallel are supported by the bidirectional switch paths. The route a message takes over the network is determined by the first 9 bits of its physical address. The number of stages in the switching network is determined by the number of ports to be supported (i.e., 2-stage switch supports 64 ports, 3-stage supports 512 ports).

A message encountering contention within the switch backs out, releasing resources until it has returned to the requestor. A rejected message is retransmitted according to a backoff algorithm [3]. After a certain number of rejections and retransmissions, the priority of a message is promoted to that of an express message, which will then be successfully delivered to its destination. Another method for controlling switch contention is a connection time limit imposed on each path. In addition to software controls, some configurations of the TC2000 switch provide alternate paths. When a conflict occurs, the message returns to its source node and is retransmitted on an alternate path.

This strategy of maintaining redundant paths prevents the message from remaining inside the switch for a long time and potentially conflicting with other incoming traffic. In one survey, Larrabee, Pennick and Stern calculate switch contention overhead to be one to five per cent of total run time, although it is application dependent. They determined that message time is normally dominated by the time required for the message to pass through the switch serially, not by contention for switching paths [1].

Similar results have been reported by various researchers at the University of Rochester on an earlier system, the GP1000. LeBlanc reported switch contention of 2% and memory contention of 3%. Experiments using four times as many memories as processors produced performance increases of 30% [9]. Mellor-Crummey also noted that increasing data locality can significantly improve performance [12].

PIC Codes

Particle-In-Cell (PIC) plasma simulations model the interaction of charged particles with electric and magnetic fields. Research problems in three dimensions often involve tens to hundreds of thousands of particles, requiring hours of CPU time on vector supercomputers. Problem size alone has motivated the search for an efficient, multi-processing solution. Previous work on PIC

plasma simulation codes on advanced architecture computers has been surveyed by Walker [14]. Development of an efficient, multi-processing solution has been slowed by the inhomogeneous nature of the problem.

The interaction of particles, which may move throughout the entire simulation space in a non-uniform manner, with field quantities maintained on a fixed spatial grid, creates the conflict in problem distribution. Each particle is defined by its position in space, velocity, mass and charge density. Electric and magnetic field quantities are discretized to each grid point. Each cycle in a PIC simulation consists of four main steps:

Assignment Phase Charge and current density for each particle, at a given position and velocity, are collected at each grid point, based on a weighting algorithm.

Field Solve Phase The electric and magnetic field equations are solved at each grid point.

Interpolation Phase Field quantities are interpolated to each particle's position, again based on a weighting algorithm.

Particle Push Phase Forces on the particles are found using the electric and magnetic fields in the Newton-Lorentz equation of motion, and used to determine the particle's new position and velocity.

Previous research has focused on problem decomposition. Lubeck and Faber [11] discuss implementation of a 2-D, electrostatic code with static decomposition of both particles and fields, on a hypercube. Problems with this approach include the communication time needed to transfer information between the divided grid of the field calculations and the replicated grid of the particle push phase. An early solution by Walker [15] was based on static decomposition with quasi-static, global communication routines. A problem with this solution is the large amount of memory required for the communication tables. Liewer, Decyk, Dawson, and Fox solve the load balance problem by using separate decompositions for particles and field quantities [10]. Two distinct spatial decompositions requires global redistribution of data twice during each time step.

For this study, a 1-D electrostatic code and a 2 1/2-D electromagnetic code were developed based on the widely used 1-D teaching code ES1 [5]. In the 1-D version, the original FFT Poisson solver was replaced with an iterative method, successive over-relaxation (SOR), to treat more general boundary conditions. Static decomposition with grid replication are used in both test codes to facilitate implementation on both distributed memory and shared memory architectures.

Initialization

For each particle, assign spatial coordinates, velocity, and contribution to overall charge density. Electric (and magnetic) fields are initialized on the grid.

Body of Simulation

particle routine

Calculate new spatial coordinates and velocity for each particle, based on current coordinates, velocity, field quantities and charge densities. Calculate new charge densities at each grid point based on the charge contributed by each particle.

field solver

Update the electric (and magnetic) fields at each grid point.

Output Final Results

Figure 2: PIC Algorithm

Implementation

Implementation of the 1-D and 2 1/2-D PIC codes was nearly identical. Both programs follow the basic algorithm shown in Figure 2. The main difference is in the solution of equations for the electric and magnetic fields. Poisson's equation is solved by successive overrelaxation (SOR) in the 1-D version. The 2 1/2-D version implements a time dependent solution to the full set of Maxwell's equations for electromagnetic fields.

The primary data structures in this implementation are shown in Table 1. For each particle, position and velocity components are stored in the particle data structure. For each grid point, electric and magnetic field components are stored in the field data structure. In addition, the grid data structure stores momentum, kinetic energy and charge density for each grid point. Data structures in this implementation consist of large (tens of thousands of elements), multi-dimensional arrays, most of which are stored in (private) common blocks.

Figure 3 illustrates the interaction of the data structures in the algorithm. Current particle data and field data are used to calculate new particle data in the particle routine. Grid data is generated for the new particle data, based on a weighting algorithm for each particle's contribution to the grid. Current field data is combined with the new grid data to calculate new values for the field data components.

Table 1: Primary Data Structures

	particle data	(for each particle)
1-D	$\langle x, v_x, v_y \rangle$	position and velocity
$2\frac{1}{2}$ -D	$\langle (x,y), v_x, v_y, v_z \rangle$	position and velocity
	field data (fo	or each grid point)
1-D	$\langle e_x, b_z \rangle$	electric and magnetic fields
2 1/2 - D	$\langle e_x, e_y, e_z \rangle$ $\langle b_x, b_y, b_z \rangle$	electric fields magnetic fields
[grid data (fo	or each grid point)
	$\langle p, ke, \rho \rangle$	momentum, kinetic energy and charge density

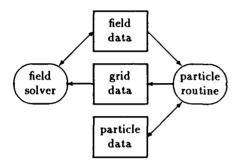


Figure 3: Data Structure Usage

In the original algorithm, the particle routine accounts for more than 90% of the computation time. Therefore, parallelization was limited to that routine. Load balancing is attained by the assignment of equal numbers of particles to each processor. The set of particles assigned to a processor may be located anywhere in the grid. Electric and magnetic field information for the entire grid must, therefore, be made available to all processors.

Temporary storage on each processor is used to localize data references. At the beginning of each time step, particle position and velocity are copied to local storage on each processor for fast access. Updated values are written to the global data structures at the end of the routine to be available for the next time step. In addition, the largest data structures are distributed in memory over all available function boards by the scatter mechanism. Distributing data is done to reduce memory contention, or hot spots, created when multiple processors are trying to access memory on a single function board concurrently.

Figure 4 details the steps in the particle algorithm. Fortran language extensions provide the constructs used for parallel programming and memory management. Synchronization, and additional processor and memory management are provided by the Uniform System li-

SUBROUTINE particle INTEGER lockvar CALL shareblk (%loc(field), fieldsize) PARALLEL REGION, REPLICATE (...) LOCAL xloc, vxloc, momentum, energy, rho CALL load (pid, xloc, vxloc) calculate new xloc, vxloc, momentum, energy, rho CALL store (pid, xloc, vxloc) CALL uslock (lockvar, 0) CALL update memory (momentum) CALL usunlock (lockvar) CALL uslock (lockvar, 0) CALL update_memory (energy) CALL usunlock (lockvar) CALL uslock (lockvar, 0) CALL update_memory (rho) CALL usunlock (lockvar) **END PARALLEL**

Figure 4: Particle Algorithm

brary routines. The PARALLEL REGION encloses a block of code which is executed once by each available processor. The REPLICATE option copies simple variables (not arrays) to each processor. The LOCAL declaration is used to create variables that are private to the PARALLEL REGION.

The particle routine begins by getting current values for particle and field data. Electric and magnetic field information is copied to each processor in a single step using the shareblk mechanism. Particle data is loaded into local storage on each processor. No synchronization is needed because each processor is loading a unique block of data (the block is identified by the pid, processor-id, variable). Local data storage is used for the calculation of momentum, kinetic energy, and current density components. The partial results from each processor are summed into the global storage for each component at the end of the particle phase. Uslock and usunlock routines enforce critical regions that prevent data corruption due to race conditions.

Results

END

The original implementation for the TC2000 using parallel FORTRAN extensions produced less than half the expected relative speedup. Additional work to op-

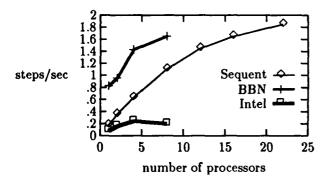


Figure 5: Steps/Second for the 1-D PIC Code

timize both processor and memory management improved speedup for a small number of processors, but did not extend successfully to larger numbers of processors. Memory management was improved by using Uniform System routines to initialize the configuration parameters and allocate space. Processor performance was improved by reducing overhead and increasing granularity.

The shareblk mechanism is an efficient way to copy read-only data to all processors. Time required to copy the large field data structure in a two-processor test was 25% of the total execution time. An experimental version eliminated the shareblk copy by using a shared, global data structure. This version increased execution time in a two-processor test by 50% and was not considered further.

Another major execution cost was the use of lock/unlock routines for synchronization. An experimental version with no lock/unlock routines executed 10-25% faster for two to eight processors than the version with locks. However, to ensure program correctness, it is necessary to replace the local data structures with a shared common block. A dimension is added to each array to store processor specific data. Summing each processor's contribution is then done outside the parallel region and protected from data corruption. Replacing local data storage with shared storage added an execution time cost of 10-25% for two to eight processors. The end result was an unchanged execution time. This version was also not considered further.

Overhead to set up the parallel region in the code includes the cost of duplicating private common blocks for every processor. It was observed that reducing the size of arrays in those blocks produced small increases in efficiency. Increasing the amount of work being done by each processor by increasing the number of particles also improved efficiency.

Figure 5 compares performance of the BBN TC2000

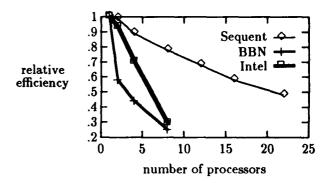


Figure 6: Relative Efficiencies for the 1-D PIC Code

with the shared memory Sequent Balance 21000 and the distributed memory Intel iPSC hypercube for the 1-D PIC code. Timing information for the 1-D shared memory version on the Balance is reported by Campbell and Sturtevant [8]. Results for the 1-D distributed memory version on the iPSC is reported by Campbell [7]. The test problem models a two-stream instability with 4096 particles in each of the two beams. The simulation executes 100 time steps over 320 grid cells. Test problems run on the TC2000 used 32768 particles in each of the two beams. The faster processors of the TC2000 required a larger granularity for efficient performance.

The number of steps/second was calculated by dividing the number of time steps by the total time, in seconds, for the problem. TC2000 steps/second were multiplied again by 65538/8192 to produce steps/second/8192 particles.

The shared memory Balance demonstrates the best performance. The number of steps/sec on the iPSC was good for a very small number of processors, but decreased rapidly for a larger number of processors. Replicating the grid on all the nodes forced global communication costs to be prohibitive. The TC2000 performance also is very good initially, decreasing as the number of processors increases.

Relative efficiences were calculated as time for 1 processor / time for N processors / N. In the graph of relative efficiences shown in Figure 6, none of the machines maintains a good relative efficiency. The Balance again demonstrates better results than both the TC2000 and iPSC.

Figure 7 compares performance of the BBN TC2000 with the shared memory Sequent Balance 21000 for the 2 1/2-D PIC code. Timing information for the 2 1/2-D shared memory version on the Sequent is reported by Campbell [6]. The test problem models a two-stream instability with 2048 particles in each of the two beams. The simulation executes 100 time steps over a grid of 10

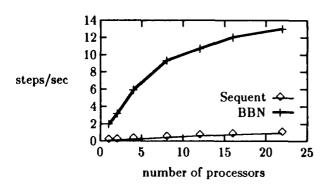


Figure 7: Steps/Second for the $2\frac{1}{2}$ -D PIC Code

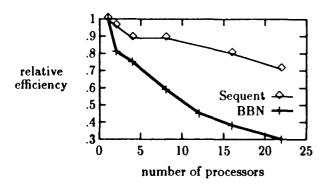


Figure 8: Relative Efficiencies for the $2\frac{1}{2}$ -D PIC Code

x 70 cells. Test problems run on the TC2000 used 32768 particles in each of the two beams for better efficiency.

The number of steps/second was calculated by dividing the number of time steps by the total time, in seconds, for the problem. TC2000 steps/second were multiplied again by 65538/4096 to produce steps/second/4096 particles.

TC2000 performance in steps/second is better than the Sequent. In the graph of relative efficiences shown in Figure 8, both machines demonstrate better efficiency than in the 1-D case. This graph illustrates the importance of relative efficiencies. The superior relative efficiency of the Balance is not demonstrated in the graph of steps/second.

Conclusions

Initial performance of each PIC code on the BBN TC2000 was somewhat below expected levels. Trivial example problems produced nearly linear, relative speedup. One such problem is the program to calculate π , originally used as a parallel test case by Babb [1]. A more complex example is a grid-based algorithm written by Dr. W. Jeffrey to solve a fluid-flow problem [4]. However, it is a very small problem (arrays with hundreds of elements), compared to the PIC codes. BBN suggests that it may help to pack code and data such that each fits into a cache. The two PIC codes implemented must be much larger than the cache size to solve physically interesting problems.

The PIC algorithm implemented was based on a shared-memory model and did not map well to the architecture of the TC2000. The high costs of copying very large blocks of read-only data and sharing very large global data structures directly affected performance. An algorithm based on a message-passing model with its natural locality of data references may be an effective solution. Communications between processors could be kept to a minimum. More information could be retained on each processor from one time step to the next, decreasing requirements for global data structures. Investigation of other models, such as message-passing, is a subject for future research.

Acknowledgments

The authors would like to thank Phil Campbell of Mission Research, Inc. and Capt. Ed Carmona of the USAF Weapons Laboratory Computational Research Group for support and encouragement. David Walker originally suggested this presentation. The Advanced Computing Research Facility at Argonne National Laboratory provided access to the BBN TC2000 used in this project. The entire support staff, in particular David

Levine, were very helpful answering technical questions. Steve Burge at BBN Advanced Computers, Inc. provided documentation.

References

- [1] Robert G. Babb II, editor. Programming Parallel Processors. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.
- [2] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. ACM Computing Surveys, 21(3), 1989.
- [3] BBN Advanced Computers Inc., Cambridge Massachusetts. *Inside the TC2000 Computer*, August 1989. Revision: Preliminary.
- [4] BBN Advanced Computers Inc., Cambridge Massachusetts. TC2000 Fortran Reference Manual, August 1989.
- [5] Charles K. Birdsall and A. Bruce Langdon. Plasma Physics Via Computer Simulation. McGraw-Hill, New York, N. Y., 1985.
- [6] Phil M. Campbell. A 2 1/2d electromagnetic pic code for multiprocessors. MRC Report MRC/ABQ-R-1241, 1990.
- [7] Phil M. Campbell. Parallel conversion of an electrostatic pic code part ii: Distributed memory. MRC Report MRC/ABQ-R-1239, 1990.
- [8] Phil M. Campbell and Judy E. Sturtevant. Parallel conversion of an electrostatic pic code part i: Shared memory. MRC Report MRC/ABQ-R-1238, 1990.
- [9] Thomas J. LeBlanc. Shared memory versus message-passing in a tightly-coupled multiprocessor: A case study. In Proceedings of the 1986 International Conference on Parallel Processing, Washington, DC, 1986. IEEE Computer Society Press.
- [10] Paulett C. Liewer, Viktor K. Decyk, John M. Dawson, and Geoffrey C. Fox. A universal concurrent algorithm for plasma particle-in-cell simulation codes. In *Proceeding of the Third Hypercube Conference*, 1988.
- [11] Olaf M. Lubeck and V. Faber. Modeling the performance of hypercubes: A case study using the particle-in-cell application. *Parallel Computing*, 9, 1988.

- [12] J. M. Mellor-Crummey. Experiences with the bbn butterfly. In Digest of Papers: COMPCON Spring 88. Thirty-Third IEEE Computer Society International Conference, Washington, DC, 1988. IEEE Computer Society Press.
- [13] Andrew S. Tanenbaum. Structured Computer Organization. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, third edition, 1990.
- [14] David W. Walker. The parallel implementation of a large-scale particle-in-cell plasma simulation code. Submitted for publication to Concurrency: Practice and Experience.
- [15] David W. Walker. The implementation of a three-dimensional pic code on a hypercube concurrent processor. In J. L. Gustafson, editor, Proceedings of the fourth conference on hypercubes, concurrent processors, and applications, 1989.

A 2D Electrostatic PIC Code for the Mark III Hypercube

R.D.Ferraro P.C.Liewer

Jet Propulsion Laboratory / California Institute of Technology 4800 Oak Grove Dr. Pasadena, CA 91109 V.K.Decyk

Department of Physics

University of California, Los Angeles

405 Hilgard Ave.

Los Angeles, CA 90024

Abstract

We have implemented a 2D electrostatic plasma particle in cell (PIC) simulation code on the Caltech/JPL Mark IIIfp Hypercube. The code simulates plasma effects by evolving in time the trajectories of thousands to millions of charged particles subject to their self-consistent fields. Each particle's position and velocity is advanced in time using a leap frog method for integrating Newton's equations of motion in electric and magnetic fields. The electric field due to these moving charged particles is calculated on a spatial grid at each time step by solving Poisson's equation in Fourier space. These two tasks represent the largest part of the computation. To obtain efficient operation on a distributed memory parallel computer, we are using the General Concurrent PIC (GCPIC) algorithm [1] previously developed for a 1D parallel PIC code.

Introduction

In previous work we have demonstrated the efficiency of a 1D PIC code on the JPL/Caltech Mark III Hypercube [1] We have now extended our work to a 2D implementation of an electrostatic PIC code for plasma simulations, using the General Concurrent PIC (GCPIC) algorithm [2]. The GCPIC algorithm is a generalization of the techniques employed in the 1D parallel PIC code which is applicable to many different parallel architectures. In this paper we describe its application to the implementation of the well benchmarked 2D electrostatic PIC code BEPSJ [3] on the Mark III Hypercube.

A plasma PIC code simulates the self consistent interactions of thousands to millions of electrons and ions in a computational box. There are two essential elements to an electrostatic PIC code. The first is the particle push, in which the positions and velocities of all of the particles are advanced in time subject to any external magnetic field and the self consistent electric, and their charges are interpolated onto the field grid. The second is the field

solve, in which the electric field is updated based upon the new particle positions. These two code sections represent the vast majority of the computation. Additional computation is required for diagnostics which are done periodically throughout the simulation, but represent an ignorable fraction of the total computation time. Thus an efficient implementation of a PIC code requires an efficient implementation of the particle push and field solve. Since the particle push represents the major fraction of the computation time, it is essential on a distributed memory machine to have approximately equal numbers of particles in each processor. The field grid must be distributed as well for the purpose of solving for the new fields, and in a manner which is not necessarily the same as that needed to push the particles. We refer to these two decompositions as the primary (particle) and secondary (field) decompositions.

Our 2D PIC code is periodic in one dimension and may be periodic or bounded in the other dimension. As the particles are advanced in time, some may traverse the entire grid space during the course of the simulation. The simplest primary (particle) decomposition which handles this problem is the static decomposition, in which each processor keeps a copy of the entire field grid and the particles are partitioned at the beginning of the simulation among processors. This technique guarantees that load balance is maintained throughout the simulation, at the expense of redundant copies of the fields in every processor. Using the static decomposition, we have obtained efficiencies for the push in excess of 80%. The major inefficiency of this method results from the need to duplicate the charge array initialization in each processor and do a sum over processors when the charge array is updated.

The next level of sophistication is to partition the field grid as well as particles, so that each processor has a unique piece of the simulation space. Because of inhomogeneity in the particle density, this partition may in general be irregular in order to maintain load balance among the processors. However, there is a large class of

2D problems which has the property of being relatively uniform along one coordinate direction, especially if the problem is periodic in that coordinate. In this case, a regular decomposition of the field grid among processors along the coordinate of uniformity (as shown in Fig. 1) will also result in a load balanced decomposition of particles. As the simulation progresses, some particles will traverse the entire simulation space. Since each processor now has only a part of the entire field grid, it is necessary to migrate particles from one processor to another as they evolve. This can result in particle load imbalance if the net flux of particles out of each processor is not zero. A particle load imbalance could develop during the course of the simulation, even though there is perfect load balance to begin with. Fortunately, for the class of problems for which this decomposition is appropriate, significant load imbalance does not develop due to the physics.

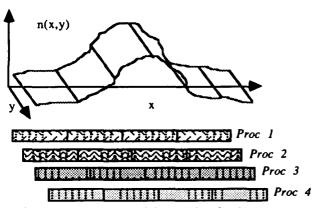


Figure 1. A regular particle partition for 4 processors. The y direction is the coordinate of relative uniformity in this case. Space is subdivided evenly among processors, leading to a load balanced partition of particles as well.

If the regular decomposition cannot be used effectively (some device physics problems can have large nonuniformities along all coordinate directions), a free form decomposition of the field grid may be necessary. These pieces can be of different size in general, since nonuniformities may develop during the course of a simulation. To maintain load balance, the distribution of particles and field grid must also evolve during the course of the simulation. We are in the process of implementing the same algorithm for dynamic load balancing as has been used for the 1D PIC code [4]. The grid space is partitioned as shown in Fig. 2 into slices so that each processor handles all of the x domain for a particular range of the y domain. Particles migrate between processors as they traverse the computational space. The grid space

may be repartitioned as the density in the simulation evolves so that load balance is maintained.

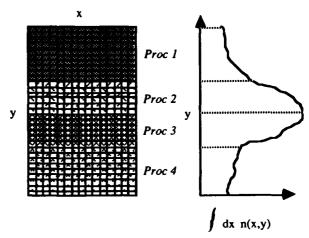


Figure 2. Field grid partition based on particle density distribution. Load balance requires that particles be distributed evenly among the processors. Thus each processor may have a different number of grid points.

The secondary (field) decomposition is made to update the field values at each time step. We calculate the new fields by solving Poisson's equation in Fourier space. For best performance in parallel, we compute the 2D FFT as two sets of 1D FFTs along each coordinate direction. For this solution method, the decomposition is a straightforward assignment of slices of the grid along one coordinate direction to each processor, as shown in Fig. 3. The

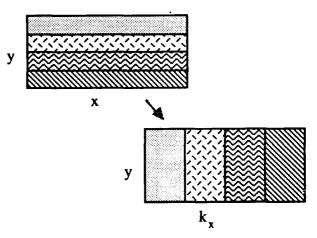


Figure 3. Field grid decomposition for the 2D FFT. Each processor has a strip of the field grid initially, such that it can do 1D FFTs in x for its subset of the y dimension. The results are then redistributed so that each processor now has a strip oriented along the y coordinate direction. 1D FFTs in y may now be performed for each processor's subset of k_x .

FFTs in the coordinate direction parallel to the long edge of the slice are performed. Then the grid is repartitioned into slices along the other coordinate direction, so that the second set of FFTs may be done.

Diagnostics are done in parallel, including graphics, by using one of the field decompositions described above. Phase space plots, for example, are parallelized using the primary decomposition, while contour plots of potential are done using the secondary field decomposition. The graphics software operates in parallel, with each processor drawing a separate portion of the graph corresponding to its part of the diagnostic.

Code Operation with the Regular Particle Decomposition

The main loop of the 2D code proceeds as follows. The field solver takes the real space charge distribution which has been interpolated onto the field grid and transforms it into k space using the 2D FFT algorithm mentioned above. Poisson's equation is solved in k space, and the x and y components of the electric field are computed from its solution. Then the two electric field components are transformed back to real space. Since the x space field grid decomposition is the same as the particle decomposition when using the regular grid primary

decomposition, no addition grid rearrangement is required to begin the push. However, interpolating the field from the grid for all of the particles in the regular decomposition requires guard rows on both sides of the grid, since particles at a decomposition boundary require field information which is contained in a neighboring processor. This guard row information is exchanged between processor neighbors before the push phase begins. By mapping the processors into a logical ring, only nearest neighbor communication is required for the exchanges. The push phase of the simulation involves advancing the particles' positions and velocities one time step, then interpolating each particle's charge back onto the field grid using its updated position. Since some particle charge will be interpolated onto the guard rows, these rows must be combined with their counterparts in adjacent processors before the charge deposition is complete. Again, only nearest neighbor communication is required.

Results

In Table 1, we present timings for the two major code section which are executed at each time step of a simulation run. Two test problems of different size were timed. In each test case, the physics problem being modeled was the same (a lower hybrid plasma wave

Timings for Critical Code Sections Mark IIIfp Hypercube

32 × 128 Field Grid

		10,12	o Particles	_		
Number of Processors	1	2	4	8	16	32
Solver (sec)	.742	.427	.275	.205	.183	Note 1
Push (sec)	1.74	.861	.421	.207	.108	Note 1
per particle (msec)	107.9	53.6	26.1	12.8	6.7	

64 × 256 Field Grid 235,136 Particles

Number of Processors	1	2	4	8	16	32
Solver (sec)	Note 2	1.70	.996	.652	.498	.449
Push (sec)	Note 2	13.2	6.66	3.34	1.68	.849
per particle (msec)	Note 2	56.1	28.3	14.2	7.1	3.6

Note 1 - The FFT requires that n_x, the number of grid points in the x direction, be at least twice the number of processors.

Note 2 - The problem was too large to fit on one processor alone.

Table 1. Measured times for the two main code sections in BEPS. Solver and Push times are elapsed times, including communication. Per particle time is push time divided by the number of particles.

traveling along the periodic coordinate was excited by an antenna). The push phase in each case shows practically linear speedup as the number of processors are increased. The solver phase, which is dominated by three 2D FFTs, rapidly saturates in speedup. This is caused by an increase in the amount of communication required by the grid redistribution between 1D FFTs while the number of 1D FFTs done in each processor decreases. This is a problem with FFT based solvers in general, since information from each grid point must ultimately be combined with information from every other grid point in order to compute the transform.

In Fig. 4 we have plotted the efficiency of each code section for the two test cases as a function of the number of processors employed. We define efficiency E as

$$E = N T_N/T_1$$

where N is the number of processors, T_N is the execution time on N processors, and T_1 is the execution time on 1 processor. The solver efficiency drops dramatically as the number of processors is increased, due to the increasing communication to computation ratio mentioned above. The push efficiency remains very near

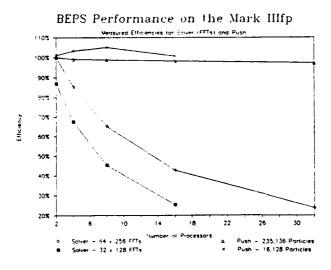


Figure 4. Measured code efficiencies. The push section of the code always runs close to 100%. The solver, which is dominated by 2D FFTs, suffers rapid efficiency degradation as the number of processors is increased.

100%, independent of the number of processors. This demonstrates that the communication time required for migrating particles between processors and exchanging guard row information is negligible compared to the

computation involved in updating the particle positions and velocities. The efficiencies in excess of 100% achieved for the smaller test case by the push phase simply indicate that the algorithm being used in the push is not optimal for one processor. The Mark IIIfp has cash memory associated with the Weitek Floating Point Processors. As more processors are used, the number of particles and the size of the field grid each processor handles decreases, resulting in a lower probability of cash misses. The increase in performance of the hardware when using the cash memory more than makes up for the addition of communication overhead. The larger test case never gets subdivided sufficiently for this hardware effect to be noticed.

Since the primary (particle) decomposition remains fixed throughout the simulation, the possibility of particle load imbalance exists. In Fig. 5 we plot the percentage of load imbalance (%LI) observed in the smaller test case running on 16 processors. The physics of the problem was changed from a heating simulation to a current drive simulation, where particles are accelerated along the periodic coordinate. This number is defined as

$$%LI = (n_{max} - n_{ave})/n_{ave}$$

where n_{max} is the maximum number of particles in any processor and n_{ave} is the average number of particles per processor. Even though particles are moving (rather

BEPS Measured Load Imbalance

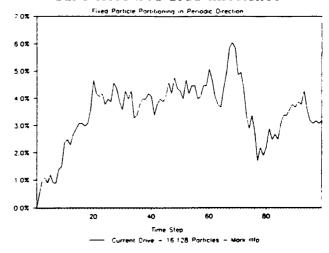


Figure 5. Measured particle load imbalance with the regular particle partition. The imbalance is defined as the largest percentage deviation of any processor's particle load from ideal at a given time step.

rapidly) between processors, the largest load imbalance observed during the first 100 time steps is about 6.2%. The load imbalance continues to oscillate around 3.5% for the rest of the simulation. This is clearly a simulation from the class where the fixed particle partition works very well. We believe, however, that the performance of the fixed particle partition on this problem is representative. Since, from a physics standpoint, it is quite difficult to develop and maintain large inhomogeneities in all coordinate directions in a plasma simulation, we also believe that the fixed particle partition is applicable to a wide variety of problems of interest to the fusion plasma and space plasma communities.

Dynamic Load Balancing

Of course not all simulation problems of interest are amenable to the fixed particle partitioning scheme. For these problems, some kind of irregular partition is necessary, and with it, the ability to dynamically balance the particle load among the processors. Fig. 6 illustrates

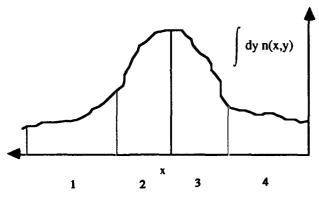


Figure 6. Dynamic load balancing without particle sorting. The charge density interpolated onto the grid is used to construct a density function. Partitioning is done based on this function.

a load balancing scheme which does not require particle sorting, per se. Assume that an irregular partition already exists which is load balanced. After the particles are advanced in time and passed among processors, some load imbalance may have developed. Rather than sorting the particles by coordinate to determine the new (load balanced) partition, the particles are interpolated onto the charge grid in the current partition. Before the field solve proceeds, the charge density is used to determine the new partition positions. The actual method of determining the new partition locations is not important, since it will scale with the grid size, rather than the number of particles. A parallel recursive bisection on the charge

density appears to be an attractive choice. We are in the process of implementing a dynamic load balancing scheme for the 2D code.

Conclusions

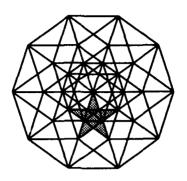
We have implemented a 2D electrostatic PIC code for plasma simulation on the Mark IIIfp Hypercube Concurrent Computer. The code is completely parallelized, including diagnostics and graphics. We are currently using a regular primary (particle) partition, which is fixed throughout the entire simulation run. This decomposition exhibits very good particle load balance for a large class of plasma problems. Particle push efficiencies remain close to 100% with up to 32 processors. Solver performance, which is based upon FFT performance, degrades rapidly as the number of processors is increased.

Acknowledgements

This work is supported by DARPA, contract # NAS7-918

References

- [1] V.K.Decyk, Supercomputer 27, 33 (1988).
- [2] P.C.Liewer and V.K.Decyk, J. Comp. Phys., 85,302 (1989)
- [3] V.K.Decyk and J.M.Dawson, J. Comp. Phys. 30, 407 (1979)
- [4] P.C.Liewer, E.W.Leaver, V.K.Decyk, and J.M.Dawson, "Concurrent PIC Codes and Dynamic Load Balancing on the JPL/Caltech Mark III Hypercube", in Proceedings of the 13th Conference on the Numerical Simulation of Plasmas



The Fifth Distributed Memory Computing Conference

18: Computational Fluid Dynamics

Massively Parallel Computation of the Euler Equations

C.E. Grosch, M. Ghose, S.N. Gupta, T.L. Jackson, and M. Zubair Old Dominion University Norfolk, Virginia 23529

Abstract

We present a systematic study of the applicability of massively parallel computers, the AMT DAP-510/310 and the TMC CM-2, to the solution of the two-dimensional unsteady Euler equations using a compact high-order scheme. The performance of these machines is compared to that of the Cray-2 and the Cray-YMP/832 using the same algorithm and for the same test problem.

Introduction

A major computational challenge is to calculate time accurate solutions to the continuum equations for the unsteady flow of compressible fluid in two and three dimensions for very large problems in a reasonable time. Computational algorithms have been developed for vector computers because, until recently, they were the only computing engines capable of providing at least a proportion of the necessary computing power. Very little work seems to have been done to develop algorithms for compressible flow calculations on a massively parallel SIMD computer. The major problem in using such massively parallel computers is to develop fine grained parallel algorithms, which requires an understanding of the data communication and synchronization requirements of these algorithms and development of efficient techniques to map the computational domain onto the set of processors.

Recently, Agarwal and Richardson [2] developed an Euler code for the TMC Connection Machine. They used a finite-volume discretization scheme coupled with a fourth-order Runge-Kutta integrator to advance the solution in time. This scheme is second-order accurate in space. In addition, since shocks can develop within the flow

field, the finite-volume method is slightly altered to allow an artificial dissipation term, which is itself second-order. The implementation of this scheme is then discussed and results were presented for a specific problem. Among the results presented, the authors showed that for large problems, the CM-2 was faster than the Cray XMP/18.

In this paper we present a systematic study of the applicability of massively parallel computers, the AMT DAP-510/610 and the CM-2, to the solution of the two-dimensional unsteady Euler equations using a compact high-order scheme. The performance of these machines is then compared to that of the Cray-2 and the Cray-YMP/832 using a vector form of the same algorithm and the same test problem.

Formulation

The two-dimensional Euler equations are,

$$\vec{u_t} + \vec{f_x} + \vec{g_u} = 0 \tag{1}$$

with

$$\vec{u} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} \tag{2}$$

$$\vec{f} = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E+p) \end{pmatrix}$$
 (3)

$$\vec{g} = \begin{pmatrix} \rho v \\ \rho v u \\ \rho v^2 + p \\ v(E+p) \end{pmatrix}$$
 (4)

where ρ , u, v, E, and p are respectively the density, velocity components in the x and y directions, the

total energy per unit volume and the pressure. In addition, the equation of state is taken to be that of a perfect gas and is given by,

$$E = p/(\gamma - 1) + \rho(u^2 + v^2)/2 \tag{5}$$

Here γ is the ratio of specific heats taken to be 1.4.

We are interested in simulating the evolution of supersonic and hypersonic mixing layers. In order to obtain accurate results in such a simulation one must control the dispersive errors and, for centered schemes, [1] these are mostly of third order hence suggesting the need for a higher than second order scheme. Thus we chose the compact four step fourth-order Runge-Kutta scheme of Abarbanel and Kumar [1]. This explicit algorithm has both fourth-order spatial and temporal accuracy as well as efficiency and ease of implementation. The Abarbanel and Kumar algorithm takes the form (where we drop the sup-arrows indicating a vector):

$$u^{0} = u^{n}$$

$$u^{1} = u^{n} - \frac{1}{4} \Delta t R(u^{0})$$

$$u^{2} = u^{n} - \frac{1}{3} \Delta t R(u^{1})$$

$$u^{3} = u^{n} - \frac{1}{2} \Delta t R(u^{2})$$

$$u^{4} = u^{n} - \Delta t R(u^{3})$$

$$u^{n+1} = u^{4}$$
(6)

where R is the compact fourth order spatial operator

$$R = \frac{\mu_x \delta_x}{\Delta x} \left(1 + \frac{1}{6} \delta_y^2\right) f + \frac{\mu_y \delta_y}{\Delta y} \left(1 + \frac{1}{6} \delta_x^2\right) g, \quad (7)$$

with δ the centered difference operator and μ the averaging operator. When shocks are present within the flow field the residual R must be modified to include an explicit artificial viscosity term which can be of second or fourth order [1].

Boundary conditions at solid walls are imposed by placing the wall between a pair of adjacent grid points. For scalar variables, such as density, pressure, etc. the boundary condition is that the gradient normal to the boundary is zero at the boundary. Thus the values of the "ghost" variables outside the flow domain are set equal to the values of the corresponding variables inside the

boundary. For vector variables, the component tangential to the boundary is treated in the same way as the scalar variables. But the boundary condition for the component normal to the boundary is that the component is zero at the boundary. This requires that the corresponding "ghost" values are the negatives of the values just within the boundary. Application of these boundary conditions requires special care in the region surrounding a convex corner. The test problems which we report on here are both supersonic. Therefore all values of the variables are set at inflow boundaries (of course these must be consistent) and no values can be set at outflow boundaries. This requires that one use extrapolation at outflow boundaries. More general boundary conditions are briefly discussed below.

Implementation

We have implemented this algorithm on a DAP-510 (at Old Dominion Univ.) and 610 (at Univ. of Cambridge) and on a CM-2 (at Argonne Natl. Lab). The DAP-510(610) consists of 1024(4096) single bit processors arranged in a 32×32 (64×64) array. Each processor is provided with connection to its four nearest neighbors. In addition, a bus system connects all the processors in each row and all the processors in each column. Each processor has a local memory of 64 Kbits.

The CM-2 can have up to 64K physical processors. These are 1-bit processors each with 64K bits of local memory. In addition, the CM-2 has one floating point processor for each set of 32 CM processors. The CM parallel instruction set provides a virtual processor facility that allows each physical processor to simulate some number of virtual processors. To transfer data among virtual processor, the instruction set supports two interprocessor communication mechanism: general communication and gridwise communication.

The DAP and CM-2 codes were written in Fortran Plus and CM-Fortran, respectively. In the case of CM-2, we used gridwise communication to transfer data among virtual processors.

The two dimensional space of the problem was divided into an equi-spaced grid. Each point of the grid is mapped onto one processor of the machine if the number of grid points is less than or equal to the number of processors. For a grid size exceeding the number of physical processors each processor acts as a virtual processor so long as

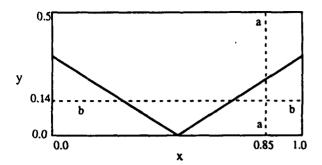


Figure 1: Shock reflection from a flat plate.

the memory size is not exceeded. Whenever an obstacle is present in the path of flow, the processors corresponding to the grid points within the obstacle are masked. That is, the values are not stored at those processors.

The implementation of this algorithm to compute the values of each component of \vec{u} at a processor consists of four similar steps, one for each step in the Runge-Kutta algorithm. Each step is in turn composed mainly of two sub-steps namely (i) acquire, and (ii) combine. In the acquire step the values of elements of \vec{f} and \vec{g} from eight neighboring processors are fetched. Four of these eight processors are directly connected to the processor concerned and hence values from those are fetched in a single step. The values from the other four processors are acquired in two steps. The eight values fetched in the acquire step are then combined to evaluate R. The artificial viscosity term is similarly computed. Subsequently \vec{u} is computed in the same way. The computation required in the acquire and combine step is done in parallel by all of the processors in the array. The boundary conditions are handled by merging rows and/or columns of data inside and outside the boundary. This is an efficient operation so that there is little deterioration in the speed of computation.

Flow Simulations

We report results for two examples: (i) a shock reflecting from a flat plate, and (ii) a Mach 3 flow into a channel with a forward facing step. Both examples are supersonic in character with shocks present, with the second example having a small recirculation region in front of the step.

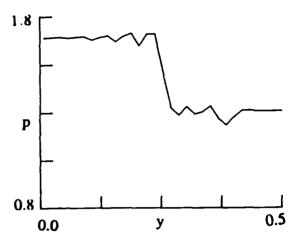


Figure 2: Pressure along section (a-a).

The geometry of problem (i) is shown in figure (1). The solid lines show the position of the shocks. The flow at the inflow, below the shock at x = 0.0, is parallel to the plate and has a Mach number of 1.95. The flow in the region above the shocks has a Mach number of 1.7736 and is inclined at an angle of 5 degrees towards the plate. This problem has an exact solution, for further details see [1] who also used this as a test problem. This test problem was run on a DAP-510 using 64 grid points in the x direction and 32 grid points in the y direction. The steady state pressure distribution along the sections (a), and (b-b) shown in figure 1 are plotted in figures 2, and 3.

The pressure distributions show that the shocks are two to three grid points thick and also that there are small oscillations near the shocks. The solution in the region to the right of the reflected shock is that given by theory. These results are very similar to those of Abarbanel and Kumar [1].

Results for the second test problem are shown in figures 4, 5, 6 and 7, where we plot contours of the density in the channel flow at four different dimensionless times, t = 0.25, 0.5, 1.0, and 2.0. Here the length scale, L, is the height of the channel at the inflow boundary and the velocity scale is c, the sound speed of the incoming gas. Thus the time scale is L/c. The impulsive start of a Mach 3 flow into a channel with a forward facing step is a severe test of the robustness of a numerical algorithm and has been used by Lohner, Morgan, and Zienkiewicz [6], and Glaister [5], for

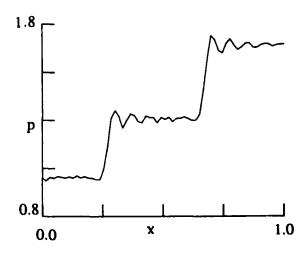


Figure 3: Pressure along section (b-b).

example, as a test problem.

The results shown here were obtained on a DAP-510. The grid for these runs was 128×32 points. At t = 0.0 a highly curved bow shock is generated at the step and begins to propagate upstream and over the step. At t = 0.25 (figure 4) the shock is curved around the step and there is a region of high density and slow circulation just ahead of the step. Notice that the shock has a thickness of about 2 to 3 grid points. By the time t = 0.50 (figure 5) the shock has moved slightly upstream of the step but has not yet hit the top wall of the channel. The shapes of the density contours in figures 4 and 5 are similar as would be expected. Figure 6 shows the density contours at t = 1.0, after the shock has undergone a reflection from the top wall. The bow shock thickness is essentially unchanged but now there is a reflected shock moving down from the upper wall. Finally, in figure 7, at t = 2.0 one can see that the shock generated by reflection at the upper wall has just begun to reflect again from the lower wall. The results of these test problem suggest that the Abarbanel-Kumar algorithm is both accurate and robust and can be used with confidence.

Performance

Here we report the performance results for the DAP-510 and 610 as well as for the CM-2, using only 8K processors, which was the maximum number available to us. We compare our timings for these machines with those obtained using

a Cray-2 and a Cray-YMP/832 to solve problem (ii). We have run the channel problem with a variety of grid sizes ranging from 96 × 32 points to 256 × 64 points on the DAP-510 and 610, on the CM-2, on a Cray-2, and on a Cray-YMP/832. Timing results are given in table 1 in terms of (a) the number of seconds required to compute one full time step for each of these grid sizes on each of these computers,(b) the relative speed of computation using a 256 × 64 grid, and (c) the processing rate.

Machine	Problem	Time	Rel.	Rate
	Size	Sec.	Speed	Mfp
CM-2	96 x 32	0.428		3.5
CM-2	256 x 64	0.474	0.56	17
DAP-510	96 x 32	0.200		7.4
DAP-510	128 x 32	0.263		7.5
DAP-510	256 x 32	0.519]	7.6
DAP-610	256 x 64	0.263	1.00	30
Cray-2	256 x 64	0.113	2.32	70
Cray-YMP	256 x 64	0.0617	4.26	128
Clay-1MIF	200 X 04	0.0017	4.20	120

Table 1: Timings and Relative Speed

From table 1 one can see that, first, there must be a major bottleneck, independent of problem size, in our CM-2 implementation. Increasing the problem size by a factor of 5.3 results in an increase in the execution time per time step of only 11 % and a consequent increase in the processing rate from 3.5 Mflops to 17 Mflops. We have not been able to find this bottleneck; perhaps because of our relative lack of experience with the CM-2. One of our major goals is to determine the cause of this bottleneck in order to see whether or not it is intrinsic to the algorithm/architecture combination.

In contrast, the DAP implementation has been very successful, perhaps because one of us (CEG) has had extensive experience in programming this, and previous, models of the DAP. The processing times and rates scale very nearly linearly with problem size and are very close to optimum; that is these rates are about 95 % of the maximum one would calculate by counting the number of floating point operations and computing the run time by multiplying the number of operations by the time per operation. The rates given in table one do not include the cost of the run

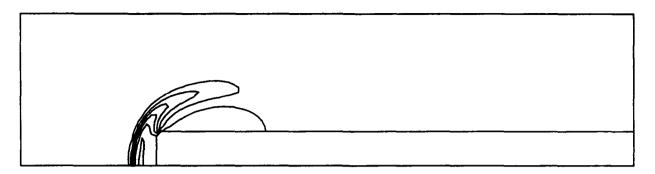


Figure 4: Density contours at t = 0.25.

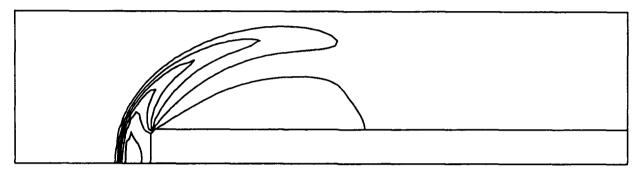


Figure 5: Density contours at t = 0.50.

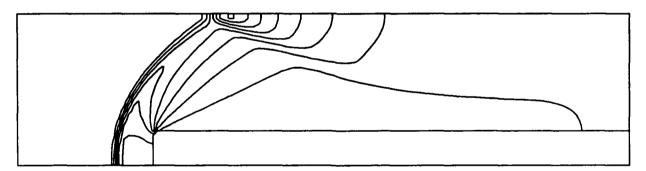


Figure 6: Density contours at t = 1.00.

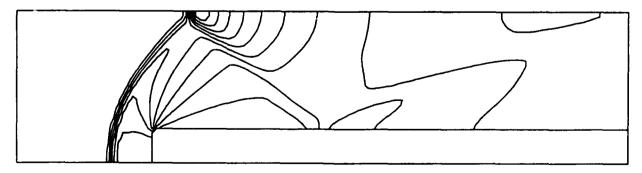


Figure 7: Density contours at t = 2.00.

time graphics where we output color contours of the density, pressure and velocity components to the monitor every time step. The cost of the run time graphics is less than 6 % of that of the time step.

The processing rates for the Cray machines, which are substantially less than the theoretical peak processing rates, are not very surprising in view of the results of our previous studies [3,4]. It seems that these rather low relative processing rates are entirely due to inadequate band width to memory and memory bank conflicts. This is a clear example of the "Von Neuman Bottleneck". It seems that massively parallel processors with a substantial local memory per processor do not experience this bottleneck.

Future Work

One of our major goals, as mentioned above, is to determine why the performance of the algorithm on the CM-2 is so disappointing. We need to find out whether or not this behavior is intrinsic to the algorithm/architecture combination or is an artifact of our implementation.

We are testing a DAP code for a direct numerical simulation of a two dimensional, spatially evolving, unstable, supersonic mixing layer on a 96 by 736 grid. We are using the Abarbanel and Kumar [1] algorithm described above. Because there are subsonic regions in the flow and the flow is unsteady, the boundary conditions must be capable of handling time dependent inflow and outflow conditions. We are now implementing characteristic inflow/outflow boundary conditions, see Thompson [7] for details.

We hope that this simulation will give us insight into the nonlinear evolution and rollup of high Mach number mixing layers. This simulation is a major computational task and will require very substantial amounts of computing time on our DAP-510. It is hoped that we can upgrade our machine to a 510C. This version, just announced, has an 8 bit coprocessor for each bit processor in the array. It is estimated that this will increase the floating point performance of the DAP by a factor of 5 to 10. Such an increase would be very welcome.

Finally, we intend to expand our study of parallel algorithms for Gas Dynamics to include TVD and ENO schemes for the Euler equations. We also will extend the algorithms so as to include general geometries using mapping techniques. Finally, we plan to develop parallel algorithms for the compressible Navier-Stokes equations suitable for massively parallel computers.

Conclusions

This study has shown that accurate and efficient finite difference algorithms for the Euler equations can be adapted to massively parallel computers. The overall performance of these codes are somewhat less than, but comparable to that of vector codes for the same algorithms on the Cray-2 and Cray-YMP. Further work to study other algorithms, general geometries, and inflow/outflow boundaries conditions seems warranted.

References

- [1] S. Abarbanel and A. Kumar, "Compact High Order Schemes for the Euler Equations", J. Sci. Comp., 3, 275-288, 1988.
- [2] R.K. Agarwal and J.L. Richardson, "Development of an Euler Code on a Connection Machine", Proceedings of the Conference on Scientific Applications of the Connection Machine, Editor:Horst D. Simon, 27-37, 1988.
- [3] R.A. Fatoohi and C.E. Grosch, "Implementation of an ADI Method on Parallel Computers", J. Sci. Comp., 2, 175-190, 1987.
- [4] R.A. Fatoohi and C.E. Grosch, "Implementation and Analysis of a Navier-Stokes Algorithm on Parallel Computers", Proceedings of the 1988 International Conference on Parallel Processing, Vol. III, D.H. Bailey, Ed., Penn. State Univ. Press, 235-242, 1988.
- [5] P. Glaister, "An Approximate Linearized Riemann Solver For the Three-Dimensional Euler Equations for Real Gases Using Operator Splitting", J. Comp. Physics, 77, 361-383, 1988.
- [6] R. Lohner, K. Morgan, and O.C. Zienkiewicz, "Adaptive Finite Element Procedure for Compressible High Speed

- Flows", Computer Methods in Appl. Mech. and Eng., 51, 441-465, 1985.
- [7] K.W. Thomson, "Time Dependent Boundary Conditions for Hyperbolic Systems", J. Comp. Physics, 68, 1-24, 1987.

Concurrent implementation of a fast vortex method

François Pépin Anthony Leonard

Graduate Aeronautical Laboratories California Institute of Technology Pasadena, CA 91125

Abstract

Vortex methods are a powerful tool for the numerical simulation of incompressible flows at high Reynolds number. They are based on a discrete representation of the vorticity field and in the inviscid limit, the computational elements, or vortices, are simply advected at the local fluid velocity. The numerical approximations transform the vorticity equation, a non-linear PDE, into a N-body problem. The $O(N^2)$ time complexity usually associated with these problems has limited the number of computational elements to a few thousands. This paper is concerned with the concurrent implementation of fast vortex methods that reduce the time complexity to $O(N\log N)$. The fast algorithm that is used combines a binary tree data structure with high order expansions for the induced velocity field. The implementation of this particular algorithm on an MIMD architecture is discussed.

Vortex Methods

Vortex methods (see Leonard[1]) are used to simulate incompressible flows at high Reynolds number. The two-dimensional inviscid vorticity equation,

$$\frac{\partial \omega}{\partial t} + \mathbf{u} \cdot \nabla \omega = 0 \quad , \tag{1}$$

is solved by discretizing the vorticity field into Lagrangian vortex particles,

$$\omega(\mathbf{x},t) = \sum_{j}^{N} \alpha_{j}(t) \, \delta\left(\mathbf{x} - \mathbf{x}_{j}(t)\right) , \qquad (2)$$

where α_j is the strength or the circulation of the j^{th} particle. For an incompressible flow, the knowledge of the vorticity is sufficient to reconstruct the velocity field. The discrete representation of the vorticity field can be used to solve

$$\nabla^2 \mathbf{u} = -\nabla \times (\omega \mathbf{e}_z) \ . \tag{3}$$

Using complex notation, the velocity induced by an isolated vortex particle,

$$\omega(z,t) = \alpha \, \delta(z - z_{\alpha}(t)) \, , \qquad (4)$$

is

$$w(z,t) = u(z,t) + iv(z,t) = \frac{i\alpha}{2\pi} \frac{1}{(z-z_{\alpha})^*} , \qquad (5)$$

where z^* is the complex conjugate of z. Since Eq.(3) is linear, superposition is used to determine that the velocity field induced by

$$\omega(z,t) = \sum_{j}^{N} \alpha_{j}(t) \, \delta\left(z - z_{j}(t)\right) , \qquad (6)$$

is given by

$$w(z,t) = \frac{i}{2\pi} \sum_{j}^{N} \frac{\alpha_{j}}{(z-z_{j})^{*}}$$
 (7)

The velocity is evaluated at each particle location and the discrete Lagrangian elements are simply advected at the local fluid velocity. In this way, the numerical scheme approximately satisfies Kelvin & Helmholtz theorems that govern the motion of vortex lines.

The numerical approximations have transformed the original partial differential equation into a set of 2N ordinary differential equations: an N-body problem. This class of problems is encountered in many fields of computational physics, e.g., molecular dynamics, gravitational interactions, plasma physics and of course, vortex dynamics. It involves a summation over (N-1) interactions that has to be evaluated N times. Even if symmetry is used to reduce the number of interactions by half, the resulting N^2 time complexity makes simulations using more than a few thousands particles prohibitively expensive.

Fast Algorithms

When each pairwise interaction is considered, distant and nearby pairs of vortices are treated with the same care. As a result, a disproportionate amount of time is spent computing the influence of distant vortices that have little influence on the velocity of a given particle. This is not to say that the far-field is to be totally ignored since the accumulation of small contributions can have a significant effect. The key element in making the velocity evaluation faster is to approximate the influence of the far-field by considering groups of vortices instead of the individual vortices themselves. When the collective influence of a distant group of vortices is to be evaluated, the very accurate representation of the group provided by its vortices can be overlooked and a cruder description that retains only its most important features can be used. These would be the group location, circulation, and possibly, some coarse approximation of its shape and vorticity distribution.

Far-field approximations

A convenient approximate representation is based on multipole expansions. Consider a compact group of J point vortices,

$$\omega_g = \sum_j^J \alpha_j \ \delta(z - z_j) \ , \tag{8}$$

where all vortices are located within a radius r_M of the group center, z_M . As discussed below, z_M is chosen in such a way to make the group as compact as possible. Other authors, like Appel [2], saw some benefits in locating z_M at the center of vorticity. In any event, the vortices induce a velocity that can be expressed as

$$w_{g}(z) = \frac{i}{2\pi} \sum_{j}^{J} \frac{\alpha_{j}}{(z^{*} - z_{j}^{*})}$$

$$= \frac{i}{2\pi} \sum_{j}^{J} \frac{\alpha_{j}}{((z^{*} - z_{M}^{*}) - (z_{j}^{*} - z_{M}^{*}))}.$$
(9)

Outside of the group, the velocity field can be rewritten as a truncated multipole expansion,

$$w_g(z) \simeq \frac{i}{2\pi} \frac{1}{(z^* - z_M^*)} \sum_{l=0}^{L-1} \frac{a_l}{(z^* - z_M^*)^l} ,$$
 (10)

which is valid for $|z - z_M| > r_M$. The coefficients a_k are defined as

$$a_{k} = \sum_{i}^{J} \alpha_{j} \left(z_{j}^{*} - z_{M}^{*} \right)^{k} \tag{11}$$

and in general, they are complex numbers. The contribution from the first neglected term drops like

$$\frac{1}{(z-z_{\scriptscriptstyle M})^L} \ . \tag{12}$$

Therefore, even a truncated series will provide an accurate velocity estimate far from z_M .

It would be possible to build a fast algorithm at this stage by evaluating the multipole expansion at the location of particles that don't belong to the group. This is basically the scheme used by Barnes & Hut [3]. Greengard & Rokhlin [4] went a step further by proposing group to group interactions. In this case, the multipole expansion is transformed into a Taylor series around the center of the second group, z_T , where the influence of the first one is sought. In the neighborhood of z_T , the induced velocity can be written as

$$w_g(z) = b_0 + b_1(z - z_T) + b_2(z - z_T)^2 + \cdots$$

$$\simeq \sum_{l=0}^{L-1} b_p (z - z_T)^l ,$$
(13)

where

$$b_{l} = \left(\frac{-1}{z_{T}^{*} - z_{M}^{*}}\right)^{l} \sum_{k=0}^{L-1} {k+l \choose l} \frac{a_{k}}{(z_{T}^{*} - z_{M}^{*})} . \tag{14}$$

An interaction between two groups consists of finding the coefficients of the Taylor series from the knowledge of the relative location of the groups and their respective multipole expansion. The work associated with this interaction is independent of the number of vortices in the groups. Consequently, the speedup over the N^2 approach is more interesting when large groups are involved. On the other hand, if the groups are small, it might be cheaper to consider every pairwise interactions between vortices. Assuming that the groups involved in the interaction have the same number of vortices, J, the critical J for which J^2 pairwise interactions of vortices require the same computational effort as one group to group interaction will be referred to as J_{\min} . No group with less than J_{\min} vortices will be allowed since they would slow down the simulation.

The threshold J_{\min} is a function of L, the number of terms in the expansions. Since the work required to compute one group to group interaction is of order $\mathcal{O}(L^2)$, it might seems preferable to keep L to a minimum but then a larger error would result from each approximation. The error, ϵ , is defined as the difference between the velocities obtained from a given group to group approximation and the ones resulting from all the pairwise interactions of the groups' members. Greengard & Rokhlin have shown that when the same number of terms is kept in both expansions, ϵ is bounded by

$$\epsilon \le A \left(\frac{r_{\text{max}}}{dr}\right)^L \quad , \tag{15}$$

where

$$dr = |z_{\scriptscriptstyle M} - z_{\scriptscriptstyle T}| , \qquad (16)$$

$$r_{\max} = \max(r_{\tau}, r_{M}) \tag{17}$$

and

$$A = \sum_{j}^{J} |\alpha_{j}| . {18}$$

These three quantities are known and an error estimate can be found for any pair of groups. If this estimate is smaller than an arbitrary criterion, $\bar{\epsilon}$, the approximation is judged acceptable and the computation can proceed with that group to group interaction. If not, at least one group is too large and the approximation is rejected since it would result in a significant error. In that case, the larger group is subdivided into two smaller ones and an error estimate is found for the two new pairs of groups. If the error is still too large, the procedure is repeated until a valid approximation is found or until the smallest groups are reached. In the latter situation, pairwise interactions between vortices are used to determine the influence of one group on another.

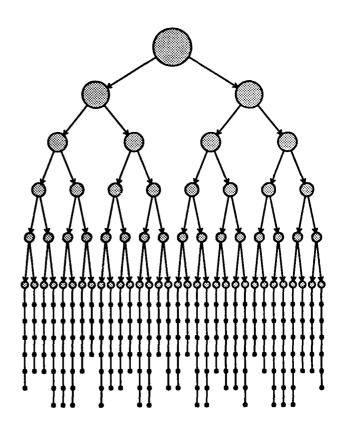


FIGURE 1 Fast algorithm's data structure

Data structure

One now needs a data structure that is going to facilitate the search for acceptable approximations. As proposed by Appel, a binary tree is used.

In that framework, a giant cluster sits on top of the data structure; it includes all the vortex particles. It stores all the information relevant to the group, i.e., its location, its radius and the coefficients of the multipole expansion. In addition, it carries the address of its two children, each of them responsible for approximately half of the vortices of the parent group. Whenever smaller groups are sought, these pointers are used to rapidly access the relevant information. The children carry the description of their own group of vortices and are themselves pointing at two smaller groups, their own children, the grand-children of the patriarchal group. More subgroups are created by equally dividing the vortices of the parent groups along the "x" and "y" axis alternatively. This splitting process stops when all groups have approximately J_{\min} vortices. Then, instead of pointing toward two smaller groups, the parent node points toward a list of vortices. As shown in Fig.(1), the data structure provides a quick way to access groups, from the largest to the smallest ones, and ultimately to the individual vortices themselves.

Velocity evaluations

Once the groups have been identified and hierarchically ordered, the coefficients of the multipole expansion that will represent everyone of them need to be evaluated. Having access to the vortices belonging to every group, Eq.(11) could be used for this purpose but it would be costly, especially for the larger groups. This expression is only used to find the coefficients of the smallest groups in the data structure, the ones that have direct access to the vortices. Then, the coefficients of the children are used to find the multipole expansion of their parent group. The expansions are constructed from the bottom up. The coefficients of the left child adequately describe its content with respect to the center of its group, z_M . To represent the left half of the parent node, that multipole expansion has to be shifted to the center of the parent node, z_{M} and the new coefficients are:

$$a'_{p} = \sum_{p=0}^{k} {p \choose k-p} a_{p} (z_{M}^{\bullet} - z_{M}^{\bullet}')^{k-p} . \tag{19}$$

The same operation is repeated for the right child and its shifted coefficients are added to the ones of the left child to form the multipole expansion of the parent group. Recursive subroutines are used to repeat this assembling process until the top of the binary tree is reached. Once the data structure is ready, the velocity evaluations can take place. The search for suitable pair of groups is done with the help of recursive subroutines, within() and between(), similar to the ones used by Appel. The subroutine between() finds the influence of one group on another while within() computes the velocities within a group. It does so by finding the interaction between its left and right halves, after which the subroutine calls itself to compute the interactions within each half. A within() of an indivisible group is simply the N^2 interaction of all its members.

When determining the mutual influence of two groups, between() first checks the error estimate associated with that pair of groups. If it is acceptable, the Taylor coefficient of each group are immediately updated. When this approximation is rejected, the largest group is split in two parts and each half interacts with the group that was not subdivided. The subroutine calls itself with smaller and smaller groups until the error estimate is small enough or until the groups cannot be subdivided anymore. In the latter case, between() does not check the error estimate but immediately proceeds with the pairwise interaction of the vortices.

Either alternative concludes the interaction of the two groups involved in the last call to between() which returns to the subroutine that called it. Before the original call to between() returns, all the between() subroutines called in the process must return as well. For the user, it appears that all velocities are computed by a single call to within(top), then the two subroutines will call themselves thousands of times until all interactions have been accounted for.

At the end of this process, some of the velocities have been directly assigned to the individual vortices but most of the information about the velocity field lies in the Taylor coefficients of the groups. Since the quantity that is updated is the location of the particles, the information accumulated in these coefficients has to be transferred downward to the appropriate vortices. The Taylor series of each group could be evaluated at all the appropriate locations but instead, shifting operations are used again. This procedure is similar to the one that took place to find the multipole coefficients with the distinction that it proceeds from the top to the bottom of the data structure. The Taylor series of the parent groups, centered around z_{τ} , are systematically shifted toward the center of their children group, z_{τ}' . The shifted coefficients are

$$b_p' = \sum_{\substack{k \\ k \ge n}}^{L} \binom{k}{p} (z_T' - z_T)^{k-p} \tag{20}$$

and are simply added to the existing ones. After they have received the contribution from their parent node, the updated coefficients are shifted downward to their own children. The process stops when the bottom of the data structure is reached; the Taylor series of the smallest groups are then evaluated at each particle location. Greengard & Rokhlin have shown that the error estimate of Eq.(15) is not affected by these shifting operations. At this point, the velocity of each vortex blobs is known and an ODE solver is used to update its location. New multipole expansions are built from the new locations and the next velocity evaluation can take place.

Appel's data structure is Lagrangian since it is built on top of the vortices and moves with them. It can be used for many time steps, but eventually, the groups will deform and could even begin to overlap. They would not be as compact as the original groups and the fast algorithm performance would deteriorate. To prevent this, the original data structure is discarded every few time steps (10 is a typical number) and new groups are identified from scratch by alternatively dividing the vortices along the "x" and "y" axis.

The data structure used by Greengard & Rokhlin is based on a spatial decomposition of the computational domain and consequently, has an Eulerian nature. The domain is subdivided into four square cells of equal area. The cells that contain more than J_{\min} vortices are subdivided again and so forth. As the vortices move, they have to be sorted again in this set of rigid boxes. This step requires little work but complicates a parallel implementation as vortices have to be exchanged between processors after each time step. On the other hand, the acceptable pairs of groups are known a priori when when a rigid data structure is used and a parallel implementation can benefit from this predictability (see Katzenelson [4]).

Fast algorithm performance

To evaluate the performance of the fast algorithm, velocities are computed for N vortices randomly distributed over a 1×1 square computational domain; their circulation is also assigned randomly. For fast algorithms based on multi-range approximations, this problem is actually a worse case scenario. When the vortex blobs are spread nearly uniformly, the groups

have to be created artificially and cannot be as compact as the ones obtained in a problem where the vortices are naturally clustered.

In any event, the velocities are first computed to double precision accuracy with the N^2 method. This is considered as the exact solution and is used as a reference against which the approximate velocities can be compared. The combination of L and $\bar{\epsilon}$ are chosen in such a way that results obtained with the fast algorithm are indistinguishable from a single precision accuracy N^2 simulation. This is a very severe restriction since the numerical integration of these velocities in time is certainly not accurate to one part in a million. However, as pointed out by Barnes & Hut, the error due to the group to group approximations could accumulate over many time steps so that one cannot allow too large an error at any given time step. In the proposed scheme, the same data structure is used for many time steps and as a result, the error vectors are correlated over a few time steps. In any event, it is preferable that the presence of the fast algorithm be as inconspicuous as possible.

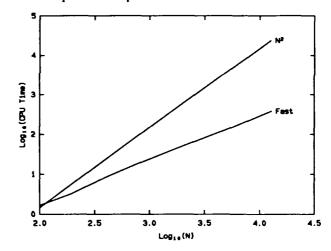


FIGURE 2 Performance of the fast algorithm

Despite this severe requirement, Fig(2) shows a remarkable speed-up over the classical approach. The CPU times are expressed in VAX 750 seconds. The crossover occurs for as few as 150 vortices; at this point, the extra cost of maintaining the data structure is balanced by the savings associated with the approximate treatment of the far field. When N is increased further, the savings outweigh the extra bookkeeping and the proposed algorithm is faster than its competitor by a margin that increases with the number of vortices. If it is clear that the computer requirement of the classical approach grows like the square

on the number of vortices, it is not as simple to determine the growth rate for the fast algorithm. Because it only involves group to vortex interactions, the Barnes & Hut scheme can be shown to be $\mathcal{O}(N \log N)$. Group to group interactions, such as presented here, remove some redundancy present in the Barnes & Hut scheme but at the same time, prevent an analysis based on the behavior of individual particles. While allowing these interactions, Greengard & Rokhlin used their rigid data structure to put an upper bound to the number of floating point operations. It was determined that their algorithm is actually $\mathcal{O}(N)$. In the proposed algorithm, the flexible data structure prevents that systematic operation count and the time complexity cannot be determined analytically but is at most $\mathcal{O}(N \log N)$. The two decades worth of data shown on Fig.(2) are not enough to determine the time complexity "experimentally". From this, one can only conclude that the difference between $O(N \log N)$ and $\mathcal{O}(N)$ makes very little difference in practice. What is really important is the constant multiplying the leading order term.

The use of recursive subroutines to search through the binary tree for acceptable interactions does not lend itself to vectorization. However, it is still true that the interactions are independent events. The influence of A on B, where A and B can be either vortices or groups of vortices, can be determined without any regard to the vorticity field that surrounds them. That inherent parallelism can be exploited to implement the method on concurrent processors.

Hypercube implementation

The fast algorithm discussed in the previous sections was implemented on the Caltech-JPL MarkIII hypercube. This MIMD machine is a Motorola 68020-based multi-processor with 4 Megabytes of memory per node. Up to 128 processors can be connected in an hypercube topology.

The quality of the parallel implementation is defined as

$$\epsilon = \frac{S}{P} \ , \tag{21}$$

the concurrent efficiency, where P is the number of processors and S is the speed-up obtained over the same application running on a single processor.

While load imbalance dominates the overhead for the concurrent fast algorithm, it is not a problem for the parallel N^2 method which is known to be very efficient (see Fox, Johnson et al. [5]). In that framework, any pair of vortices represents the same amount of work and the load can be perfectly balanced by assigning the same number of vortices to each processor. Furthermore, the domain decomposition can be done without paying any attention to the location of the vortices. To find the velocities, each processor makes a copy of its vortices and sends it to half of the other processors where it interacts with the resident vortices. The contributions to the velocities of the visiting copy are accumulated as it is sent from processor to processor. Ultimately, it is sent back to its original processor where these contributions are added to those of the copy that stayed there. A large amount of data has to be exchanged between processors but this application is so computer intensive that the time spent computing velocities dwarfs the communication time and efficiencies close to unity can be achieved for large problems. The regularity of the problem also allows a synchronous implementation which further reduces the time spent communicating between the nodes.

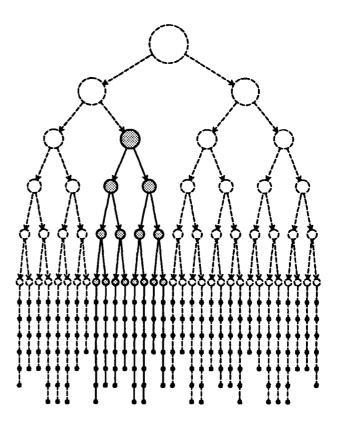


FIGURE 3 Data structure assigned to processor 1

The global nature of the N^2 approach has made its parallel implementation fairly straightforward. However, that character was drastically changed by the fast algorithm as it introduced a strong component of locality. Globality is still present since the influence of particle is felt throughout the domain, but more care and computational effort is given to its near field. The fast parallel algorithm has to reflect that dual nature, otherwise an efficient implementation will never be obtained. Moreover, the domain decomposition can no longer ignore the spatial distribution of the vortices. Nearby vortices are strongly coupled computationally. Hence, it makes sense to assign them to the same processor. The binary tree data structure could be used for that purpose. By dismissing the (P-1) largest groups in the data structure, P groups containing approximately the same number of vortex blobs can be identified and a different processor is assigned the responsibility of each of these subtrees. For example, Fig.(3) shows the portion of the data structure assigned to processor 1 in a four processor environment.

This strategy ensures that the vortices given to each processor are actually neighbors in the physical space. The drawback of this approach is that the full data structure has to be constructed in the host processor before portions of it can be sent to the hypercube. In practice, binary bisection is used in the host to spatially decompose the domain. Then, only the vortices are sent to the processors where a binary tree is locally built on top of them. Less data has to be loaded on the hypercube and the generation of the local binary trees can be done in parallel.

In a fast algorithm context, sending a copy of local data structure to half the other processors does not necessarily result in a load balanced implementation. The work associated with processor to processor interactions now depends on their respective location in physical space. Besides, a processor whose vortices are located at the center of the domain is involved in more costly interactions than a peripheral processor. To achieve the best possible load balancing, that central processor could send a copy of its data to more than half of the other processors and hence, be itself responsible for a smaller fraction of the work associated with its vortices.

Before a decision is made on which one is going to visit and which one is going to receive, the number of pairs of processors that need to exchange their data structure needs to be minimized. Following the domain decomposition, the portion of the data structure that sits above the subtrees is not present anywhere in the hypercube. That gap is filled using recursive doubling to make the description of the largest group of every processor known to everybody else. By limiting the broadcast to one group per processor, a small amount of data is actually exchanged but, as seen on Fig.(4), this step gives every processor a coarse description of its surroundings and helps it find its place in the universe.

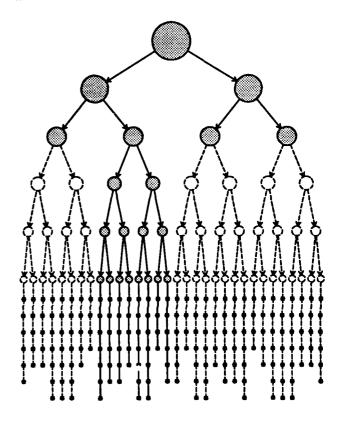


FIGURE 4 Data structure known to processor 1 after broadcast

If the vortices of processor A are far enough from those of processor B, it is even possible to use that coarse description to compute the interaction of A and B without an additional exchange of information. The far field of every processor can be quickly disposed of. After thinking globally, one now has to act locally; if the vortices of A are adjacent to those of B, a more detailed description of their vorticity field is needed to compute their mutual influence.

This requires a transfer of information from either A to B or from B to A. In the latter case, most of the work involved in the A-B interaction takes place in processor A. Obviously, processor B should not always

send its information away since it would then remains idle while the rest of the hypercube is working. Load balancing concerns will dictate the flow of information. To do so, a list of all the interactions requiring a further data exchange is drawn in every processor. Since the upper portion of the tree has been duplicated P times, an identical copy of that list is created simultaneously in every processor. Then the responsibility of each item in the list is assigned to either processors involved while trying to distribute the resulting computational load as equally as possible.

Since vortices move only slightly during each time step, the computational work required for the interaction of two given processors at the previous time step can be used as an estimate of the work involved for the present one. The pairs in the list are examined sequentially; the processor with the lightest work load when the pair is considered is given the responsibility of the interaction and computes the interaction after receiving the data structure from the other one. The work load that is used to make that decision is the sum of the work estimates already assigned to the processor plus half of the estimates of the interactions in which that processor is involved but have yet to be assigned. Ultimately, every processor knows not only where to send its data but also from which processor it should expect to receive additional information. The latter will be referred to as the request list of a processor.

The first round of communication can now take place. To ensure that processors are not overloaded with data, information is sent upon request only. Each processor first checks if it is at the top of the request list of any other processors. If so, it immediately sends a copy of its data structure to the proper recipient(s). Every processor receives one and only one visiting data structure. As soon as it arrives, this structure interacts with the local groups and vortices. Upon completion of that operation, the processor which was responsible for the interaction sends a message to the next processor in its own request list to let that processor know that a copy of its data is now needed at a specific location. The updated velocities and Taylor series coefficients of the visitor are also sent back to their origin where they are added to the local data structure. Processors frequently peek at their message queue to make sure that requests get an immediate answer and that the returning information is absorbed as quickly as possible. This keeps the message queue to a manageable size. The processor that has just sent the request then has to wait for the arrival of the next visitor.

To reduce the idle time, more than one request

can be filled at the beginning of the process creating a stack of visitors in every processor. Each processor receives two or three visitors and gets to work as soon as the first one arrives; the other ones are left in the stack. When the first interaction is completed and the next request sent, a processor can already start working on the next visitor in its stack. Memory restrictions limit the stack size to two or three visiting data structures. When all visiting copies have returned to their origin, the processors consider the interactions among their own vortices. Then, the vortices location and the whole data structure are updated. The process starts over again by broadcasting the largest group of each processor.

Obviously, this message sending takes place asynchronously. Furthermore, the MarkIII is considered as a collection of computers loosely connected through an arbitrary network; the hypercube topology is not used as such.

At this point, it should be noted that the data structure used in a parallel implementation differs significantly from the one used on a sequential computer. In the latter case, the parent group points toward his children using memory addresses. On concurrent computers, the local binary trees are exchanged between processors and addresses that were valid where the tree was constructed are meaningless in a different processor.

Instead, the data structure is built inside a one dimensional array and parent groups refer to their children by their indices. Two arrays are actually used, one for the vortices, $V[\]$, and one for the groups, $G[\]$. When additional information is requested, $G[\]$ is sent immediately; then the respective location of the processors vortices is considered to determine if $V[\]$ should follow. If the processors are adjacent, the full description of the vorticity field is needed but if they are sufficiently far away, the description provided by the groups is adequate and $V[\]$ can stay home.

Efficiency of parallel implementation

Since our objective is to compute the flow around a cylinder, the efficiency of the parallel implementation was tested on such a problem. The region for which 1 < r < 1.6 is uniformly covered with N particles. The parallel efficiency, as defined in Eq.(21), is shown on Fig.(5) as a function of the hypercube size. The parallel implementation is fairly robust as ϵ remains larger than 0.7 for a 32-node concurrent computer meaning that a typical processor does useful work at least 70%

of the time. The number of vortices per processor was kept roughly constant at 1500 even if the parallel efficiency is not a strong function of the problem size.

It is, however, much more sensitive to the quality of the domain decomposition. The fast parallel algorithm performs better when all the sub-domains have approximately the same squarish shape or in other words, when the largest group assigned to a processor is as compact as possible.

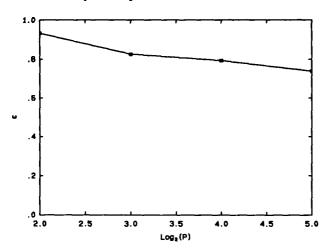


FIGURE 5 Parallel efficiency of the fast algorithm.

The results of Fig.(5) were obtained at early times when the Lagrangian particles are still distributed evenly around the cylinder which makes the domain decomposition an easier task. At later times, the distribution of the vortices does not allow the decomposition of the domain into P groups having approximately the same radius and the same number of vortices. Some subdomains cover a larger region of space and as a result, the efficiency drops to approximately 0.6. This is mainly due to the fact that more processors end up in the near field of a processor responsible for a large group; the request lists are longer and more data has to be moved between processors.

The sources of overhead corresponding to Fig.(5) are shown on Fig.(6) normalized with the useful work. Load imbalance, the largest overhead contributor, is defined as the difference between the maximum useful work reported by a processor and the average useful work per processor. It is a measure of how much faster the simulation would have been if the load had been equally divided among the processors. Secondly, the extra work includes the time spent making a copy of one's own data structure, the time required to absorb

the returning information and the work that was duplicated in all processors, namely, the search for acceptable interactions in the upper portion of the tree and the subsequent creation of the request lists. The remaining overhead has been lumped under communication time although most of it is probably idle time (or synchronization time) that was not included in the definition of load imbalance.

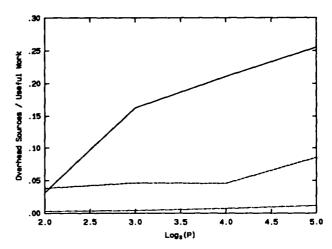


FIGURE 6 Load imbalance (solid), communication & synchronization time (dash) and extra work (dot-dash) as a function of the number of processors.

It was originally expected that as P increases, the near field of a processor would eventually contain a fixed number of neighboring processors. The length of the request lists and the load imbalance would then reach an asymptote and the loss of efficiency would be driven by the much smaller communication and extra times. However, this has yet to happen at 32 processors and the communication time is already starting to make an impact. Nevertheless, the fast algorithm, its reasonably efficient parallel implementation and the speed of the MarkIII have made possible simulations with as many as 80,000 vortex particles.

Acknowledgments

The authors gracefully acknowledge the support of fonds F.C.A.R., NSERC and the Department of Energy under the grant #DE-FG03-85ER25009. We also wish to thank Dr Paul Messina for the access to the Caltech Concurrent Supercomputing Facilities.

References

- [1] Leonard, A. 1980. Vortex methods for flow simulation. J. Comput. Phys. 37, 289.
- [2] Appel, A. 1985. An efficient program for manybody simulation. SIAM J. Sci. Stat. Comput. 6, 85.
- [3] Barnes, J. & Hut, P. 1986. A hierarchical O(N log N) force-calculation algorithm. Nature 324, 446.
- [4] Greengard, L. & Rokhlin, V. 1987. A fast algorithm for particle simulations. J. Comput. Phys. 73, 325.
- [5] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. & Walker, D. 1988. Solving problems on concurrent processors. Prentice-Hall.
- [6] Katzenelson, J. 1989. Computational structure of the N-body problem. SIAM J. Sci. Stat. Comput. 11, 787.

Parallel Computation of the Compressible Navier-Stokes Equations with a Pressure-Correction Algorithm

Mark E. Braaten
GE Research and Development Center
P. O. Box 8
Schenectady, NY 12301

Abstract

A parallel algorithm for the solution of the 2D compressible Navier-Stokes equations has been developed and demonstrated on a distributed memory multicomputer. The algorithm represents an extension of an earlier parallel incompressible pressure correction algorithm developed by the author. The new algorithm features a revised formulation of the pressure correction equation that simultaneously updates both velocity and density to enforce continuity, and uses upwinding of the densities to allow shock capturing. The parallel implementation is based on a full two-dimensional domain decomposition. As in the earlier algorithm, an effective block correction procedure is found to be the key to high parallel efficiency. Results are obtained on a 32-node Intel iPSC/2VX hypercube for inviscid and viscous transonic flows in turbomachinery blade rows. Performance approaching 1/4 that of a single-processor Cray Y-MP is achieved.

Introduction

Traditionally, different methods have been used to solve compressible and incompressible flows. Time-marching methods such as Jameson's explicit Runge-Kutta scheme [1] and the Beam-Warming implicit scheme [2] are commonly used for the solution of compressible flows. These methods treat the continuity equation as an equation for the density, and then extract the static pressure from the equation of state. Such methods fail in the incompressible limit of zero Mach number since the density becomes independent of the pressure, and the pressure cannot be calculated from the density. Pressure correction methods, in which the pressure is solved for directly, rather than the density, have proven highly successful for incompressible flows [3]. The pressure is calculated via an equation for pressure corrections which is derived by algebraic manipulations of the discrete momentum and continuity equations.

In recent years, the pressure correction formulation has been extended to handle compressible flow (see, for example [4,5]). The resulting algorithm has the attractive property of being able to address inviscid, laminar, and turbulent flows at all Mach numbers, making it very widely applicable.

In some earlier papers [6,7], the present author described the development of a parallel pressure correction algorithm for laminar and turbulent incompressible flows, and its implementation on a distributed memory multicomputer. The parallel algorithm was based on a stripwise domain decomposition, and the development of an effective parallel block correction procedure which eliminated the convergence penalty caused by the domain decomposition. Speedups in excess of 20 were achieved with 32 scalar processors on an Intel iPSC/2, and performance with 8 vector processors using an Intel iPSC/2VX approached 1/5th that of a single processor Cray-XMP.

The work described in this paper represents an extension of the earlier algorithm to a parallel compressible pressure correction algorithm applicable at all Mach numbers. The original stripwise domain decomposition has been replaced with a full two-dimensional decomposition, to allow for the use of more processors. The paper focuses on the highlights of the compressible formulation and the implementation of the block correction procedure on a two-dimensional mesh of processors. Finally the performance of the algorithm is demonstrated for two test calculations involving inviscid and viscous transonic flow in a turbomachinery blade row.

Highlights of the Compressible Formulation

The compressible pressure correction algorithm developed here solves the two-dimensional Navier-Stokes equations for viscous flow or the Euler equations for inviscid flow. The equations for conservation of x-momentum, y-momentum, and mass are solved, along with the equation of state. For now,

the temperature has been calculated from the assumption of constant stagnation enthalpy, rather than by solving the compressible form of the energy equation. For turbulent flows, the standard k- ε turbulence model is used, along with the wall function treatment for the near-wall regions [8].

The governing equations are expressed in Cartesian coordinates, and then transformed to a general body-fitted coordinate system $\xi = \xi(x,y)$, $\eta = \eta(x,y)$. In the interest of space, the reader is referred to References 9 for complete details.

In the pressure correction approach, the momentum equations are first solved with a guessed pressure field p^{\bullet} . The resulting velocity field does not necessarily satisfy continuity. The pressure correction equation is obtained by substituting simplified forms of the momentum equations into the continuity equation to obtain an equation for the pressure correction p, defined such that the corrected pressure p is given by

$$p = p^{\bullet} + p^{'} \tag{1}$$

Once the pressure corrections are obtained from the solution of this equation, the pressure is updated and the velocities are corrected to satisfy continuity.

The distinction between the incompressible and compressible pressure correction algorithms stems from the fact that the density is taken as fixed during the course of the pressure correction in the incompressible algorithm, while it is taken to be a function of pressure in the compressible algorithm [5]. A density correction ρ' is defined such that

$$\rho' = c^{\rho} \rho' \tag{2}$$

The mass flux terms in the discrete continuity equation can then be decomposed into four parts, i.e.

$$\rho U = \rho^* U^* + \rho^* U^{'} + \rho^{'} U^* + \rho^{'} U^{'}$$
 (3)

The first two terms, which are the only terms present in the incompressible form, represent the mass flux calculated from the given density and velocity fields, and the contributions from the velocity corrections. The last two terms are the contributions arising from compressibility, representing the linear contribution from the density corrections, and the nonlinear contribution from the compressibility effect, respectively. In contrast to earlier works, the

nonlinear term is retained, since it helps to stabilize the procedure in the early iterations when neither ρ or U is necessarily small. The nonlinear terms are lagged during the course of the iterative solution of the pressure correction equation at each iteration of the overall procedure.

The addition of the compressible terms results in a nonlinear convection-diffusion equation for the pressure corrections, rather than the linear diffusion equation obtained in the incompressible algorithm. Experience has shown that the compressible pressure correction equation is more difficult to solve than its incompressible cousin, and that close enforcement of continuity at each iteration is even more crucial for success of the compressible algorithm than in the incompressible case.

Upwinding of the densities provides the mechanism for shock capturing in transonic and supersonic flows. Although the first-order accurate hybrid differencing scheme is still widely used in incompressible flows, it leads to excessive smearing when shocks are present and excessive total pressure errors. In this work, the hybrid scheme was replaced by the conservative second-order accurate QUICK [10] scheme. QUICK was found to capture shocks within 3-4 grid cells and lead to significantly better total pressure conservation in inviscid flows.

Parallel Implementation

The basic parallel implementation of the pressure correction algorithm using a stripwise decomposition was described in detail in [6,7], and will not be repeated here. In this work, a full two-dimensional decomposition was used. A 2D decomposition has the advantage that it allows the use of more processors than a 1D decomposition, which is limited to a number of processors no greater than the number of cells in any one direction. The major impact of using the 2D decomposition on the parallel algorithm was on the implementation of the block correction procedure, and on obtaining good performance from the vector processors.

The two-dimensional domain decomposition is done by subdividing the solution domain into overlapping rectangular regions (in the transformed space). Overlapping of the solution domains is required so that each interior cell is computed as an interior point by at least one processor. An overlap of one cell in each direction at each face was used, since this minimizes the redundant storage of quantities in the overlapping regions. The use of a one cell overlap in conjuction with the second-order QUICK scheme requires some special treatment,

since each cell requires information from two cells upstream when QUICK is used. Although the simplest way to implement QUICK would be to use a two cell overlap on each edge, this leads to very inefficient memory utilization as the number of processors gets large (for a problem of fixed size). The approach adopted here was to utilize four temporary vectors, one for each interface, to store the extra u and v values for the cells adjacent to the interface. Since QUICK is used only in the discretization of the x- and y- momentum equations, the only penalty is the cost of passing four additional messages for each momentum equation and the storage required by the four temporary vectors, both of which are minimal.

A key finding in the earlier work [7] was that an efficient parallel block correction procedure was the key to maintaining high parallel efficiency as the number of processors was increased. The block corrections eliminate the reduction of convergence rate of the elliptic pressure correction equation that occurs when the solution is done by a parallel Schwarz alternating method, rather than by an implicit solution over the entire domain. In this work, the same block correction procedure was implemented, and again found to be crucial to achieving convergence rates essentially independent of the number of processors. The switch to a twodimensional decomposition from the earlier stripwise decomposition necessitated significant changes in its implementation, which are described in the following paragraphs.

In the block correction procedure [11], a series of one dimensional corrections are made to the solution of the pressure correction equation, first over rows of the grid, and then over columns. The following discussion will focus on the corrections on the columns; the row corrections follow similarly. The coefficients for the column corrections are obtained by summing the original discretized coefficients across each column. Since in a 2D decomposition, no processor spans an entire column of grid cells, this summation step requires communication among the processors in a given column. Notice that in the original stripwise decomposition reported earlier, the processors did span entire columns, and this extra step was not needed. Once the coefficients for each row are summed, they need to be are exchanged among all of the processors, so that each processor can compute the corrections independently. The resulting correction equations are of tri-diagonal form, and can easily solved using the tri-diagonal matrix algorithm (TDMA). With the stripwise decomposition, the exchange

coefficients was done over all of the processors. With the 2D decomposition, each processor needs only to exchange coefficients with the other processors that share the same row of grid cells.

The use of binary reflected gray code (BRGC) to map the processors onto a two-dimensional mesh leads to the most efficient communication not only for the local communication between neighboring processors, but also for the rowwise and columnwise exchanges of information required by the block correction procedure. Figure 1 shows a BRGC mapping for a 4 x 4 mesh of processors on a fourdimensional hypercube. Note that not only are all of the neighboring processors nearest neighbors (their binary node numbers differ in only one digit), but also that the numbers of all of the processors in a given row share the same first two digits, and all of the procesors in a given column share the same last two digits. This means that both the columnwise and rowwise exchanges required by the block correction can be performed via global shuffles and global concatenation operations on subcubes of dimension 2. This is not the case if the processors are mapped onto the mesh in simple lexicographical order.

Another important consideration in the use of a 2D decomposition on a multicomputer with vector processors is the need to maintain sufficiently long vector lengths for good vector performance. The vector length required to reach one-half of the peak performance (the so-called half length) is on the order of 50 words on the iPSC/2VX vector processor [7]. If two-dimensional data structures are used in the code (i.e. A is stored as A(I,J)) then the resulting code will contained nested DO loops over I and J. For such cases, only the innermost DO loop will vectorize. For a 128 x 64 mesh on an 8 x 4 mesh of processors, each processor treats a 16 x 16 problem. The resulting vector length will be only 16, far below the half length, and vector performance will be poor. The use of one-dimensional data structures (i.e. A is stored as A(IJ), where IJ = (I-1)*NJ+J) results in a vector length of 256 for the same loop, and near-peak vector performance will be achieved. For this reason, the code developed here uses a one-dimensional data structure throughout, and utilizes a single loop over all of the grid cells whenever possible.

One final note regards the choice of the iterative solver used to solve the linearized equations at each stage of the algorithm. Two methods have been implemented, namely line-by-line TDMA and a vectorized point-symmetric Gauss-Scidel method [7].

The line-by-line TDMA is found to give faster convergence due to its superior performance on the compressible pressure correction equation. However, the vectorized point solver leads to better vector performance, resulting in a significantly lower cost per iteration, at the price of somewhat slower convergence. Since the optimum choice appears to be problem dependent, both methods have been retained.

Test Results

The parallel pressure correction algorithm described here has been tested on a number of subsonic, transonic, and supersonic flows in inlets, channels, and blade row cascades. Inviscid, laminar, and turbulent flows have been computed. Due to space limitations, this section will focus on two calculations for transonic flow in a hypothetical turbomachinery cascade. The first case solves for the inviscid flow through the cascade using the Euler equations, while the second case solves for turbulent flow in the same cascade using the Navier-Stokes equations plus the k- ε turbulence model. The inlet Mach number is 0.7, and the flow enters the cascade at a 35° inlet angle. The computations were performed on an H-grid with 128 x 64 grid cells; a closeup view of the mesh in the vicinity of the leading edge is shown in Figure 2. The mesh was constructed from a 112 x 48 mesh by subdiving a number of the original cells near the blade surfaces and also by adding grid lines in the anticipated vicinity of the shock.

The performance of the parallel algorithm was explored by running 100 iterations of the algorithm on both the scalar and vector processors of a 32node Intel iPSC2/VX. The results for the cpu time and parallel efficiency are shown in Figures 3 and 4. With 8 vector processors, the code runs 1.5 times faster with the line solver, and 3.2 times faster with the point solver, than the code with the line solver on 8 scalar processors. With 32 vector processors, the corresponding values both fall to 1.4, due primarily to a reduction in vector lengths as the number of processors increases. The parallel efficiency, estimated based on the procedure described in [7], is found to be reasonably high. With 32 processors, efficiencies of 80% are achieved with scalar processors, 71% with the line solver on vector processors, and 45% for the point solver on vector processors. Again the shorter vector lengths that occur when more processors are used are a major factor in reducing the parallel efficiency for the vector hypercube, particularly when the vectorized point solver is used.

A highly vectorized version of the original serial code was run on a single processor of a Cray Y-MP. The cpu times for 100 iterations were 60.7 seconds for the line-by-line TDMA, and 30.0 seconds for the vectorized point solver. Hence, the performance of the 32-node hypercube was about 1/4th that of the Cray for the line solver, and about 1/5th for the point solver.

In a transonic blade row of this type, the blockage due to the thickness of the viscous boundary layer can have a significant effect on the shock location and on the resulting total pressure losses. Figure 5 shows the computed Mach number and pressure contours for the inviscid test problem. Figure 6 shows the corresponding results for the turbulent test problem. Reasonably converged solutions were obtained in 1500 iterations in both cases. The presence of a shock upstream of the trailing edge of the blade is clearly evident. The turbulent solution correctly predicts the influence of the boundary layer blockage on the shock location, as the shock clearly moves upstream from its inviscid location. The interaction of the shock with the boundary layer causes the boundary layer to separate, leading to a large region of recirculation near the trailing edge in the turbulent case.

An interesting result that is observed in the inviscid solution regards the apparent glitches in the Mach number contours. The location of these glitches corresponds to the regions of the grid where the cell spacing varies abruptly by a factor of two as a result of the manner in which the final grid was constructed by adding grid lines to the original grid. The second-order QUICK scheme responds to the abrupt change in spacing by wiggling. Notice that the viscous solution shows no such behavior; evidently there is enough physical dissipation in the viscous equations to suppress the wiggles. The wiggles in the inviscid case are evidence that the level of numerical dissipation in the QUICK scheme is quite low. Since any wakes observed in the inviscid solution are also evidence of numerical dissipation, the small size of the inviscid wake is further evidence of the low level of dissipation in the QUICK scheme. The lesson here is that inviscid test problems provide a demanding challenge for any numerical formulation, and therefore are very useful for exploring the accuracy of the formulation, even if the ultimate goal is to analyze viscous flows.

Concluding Remarks

A parallel compressible pressure correction algorithm applicable at all Mach numbers has been

developed and demonstrated on a distributed memory multicomputer. Reasonable high parallel efficiencies have been achieved, and performance up to 1/4th that of a single Cray Y-MP processor has been obtained. Calculations of turbulent transonic flow in a turbomachinery cascade correctly predict the effect of the boundary layer blockage on the shock position and the resulting losses. With the advent of the next generation of multicomputers, with faster processors (such as the Intel i860), and faster interprocessor communication, this parallel algorithm should achieve performance beyond that of existing serial algorithms on conventional supercomputers and allow larger and more accurate simulations to be performed.

References

- [1] A. Jameson, W. Schmidt, and E. Turkel, "Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time Stepping Schemes", AIAA paper AIAA-81-1259, 1981.
- [2] R. W. Beam and R. F. Warming, "An Implicit Finite Difference Algorithm for Hyperbolic System in Conservation Form", J. Comp. Phys., vol. 23, pp. 87-110, 1976.
- [3] S. Patankar, Numerical Heat Transfer and Fluid Flow, Hemisphere, Washington, DC, 1980.
- [4] C. Rhie, "A Pressure Based Navier-Stokes Solver Using a Multigrid Method", AIAA Paper 86-0207, 1986.
- [5] W. Shyy and M. E. Braaten, "Adaptive Grid Computation for Inviscid Compressible Flows Using a Pressure Correction Method", AIAA paper 88-3566-CP, 1988.
- [6] M. E. Braaten, "Solution of Viscous Fluid Flows on a Distributed Memory Concurrent Computer", Int. J. Num. Meth. Fluids, in press, 1990.
- [7] M. E. Braaten, "Development of a Parallel Computational Fluid Dynamics Algorithm on a Hypercube Computer", GE Research and Development Center Report 89CRD030, May 1989. Submitted to Int. J. Num. Meth. Fluids, 1989.
- [8] B. E. Launder and D. B. Spalding, "The Numerical Computation of Turbulent Flows", Comput. Meth. Appl. Mech. Engrg., vol. 19, pp. 59-98, 1974.

- [9] M. E. Braaten and W. Shyy, "A Study of Recirculating Flow Computation using Body-Fitted Coordinates: Consistency Aspects and Mesh Skewness", Num. Heat Transfer, vol. 9, pp 559-574, 1986.
- [10] B. P. Leonard, "A Stable and Accurate Convective Modeling Procedure Based on Quadratic Upstream Interpolation", Comput. Meths. Appl. Mech. Eng., 19,pp. 59-98, 1979.
- [11] A. Settari and K. Aziz, "A Generalization of the Additive Correction Methods for the Iterative Solution of Matrix Equations", SIAM J. Numer. Anal., vol. 10, pp 506-521, 1973.

10 00	10 01	10 11	10 10
11 00	11 01	11 11	11 10
01 00	01 01	01 11	01 10
0000	00 01	0011	00 10

Fig. 1. Gray code mapping for 4 x 4 processor mesh

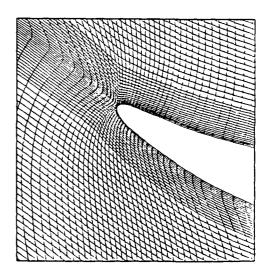


Fig. 2. Closeup of grid near blade leading edge

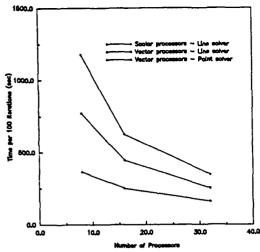


Fig. 3. Performance of parallel algorithm - 100 iterations, inviscid test case

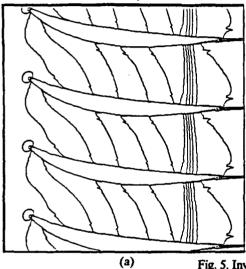


Fig. 5. Inviscid transonic cascade test case

- (a) Mach number contours
- (b) Pressure contours

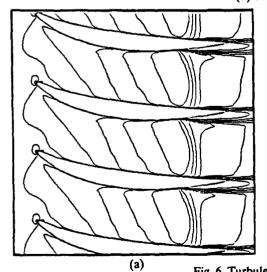


Fig. 6. Turbulent transonic cascade test case

- (a) Mach number contours
- (b) Pressure contours

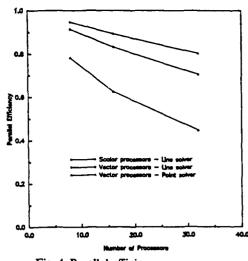
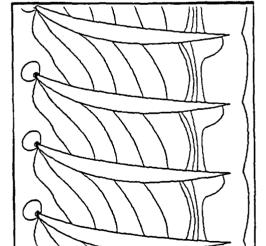
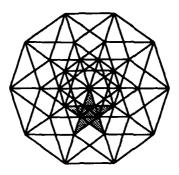


Fig. 4. Parallel efficiency

- 100 iterations, inviscid test case



(b)



The Fifth Distributed Memory Computing Conference

19: Other Scientific Applications

Hypercube Simulation of Electric Fish Potentials

Roy Williams, Brian Rasnow and Christopher Assad

California Institute of Technology
Pasadena, California

Abstract

We present a simulation of the electrosensory input of the weakly electric fish Apteronotus leptorhynchus. This fish senses its environment by producing a sinusoidal voltage difference between its body and tail sections, causing an electric field and a current distribution in the surrounding water. If an object is nearby which has different electrical conductivity from the surrounding water, the current distribution is disturbed on the skin of the fish. The fish senses this difference from the usual current distribution, and infers the presence and location of the object.

Mathematically, the problem is to solve a potential equation in the domain exterior to the fish with Cauchy boundary conditions, in the presence of an induced dipole arising

from the object, and extract the potential difference across the fish skin.

We have created an unstructured triangular mesh covering the two-dimensional manifold of the fish skin, using the Distributed Irregular Mesh Environment (DIME), then used the Boundary Element Method to solve for the potential derivative at the fish skin.

The computational problem is the solution of a full set of simultaneous linear equations, where there is an equation for each node of the boundary mesh, typically about 100 - 200. We have used an NCUBE hypercube to calculate the matrix elements and solve these equations, once for each relative position of the fish and the test object. We present some early results from the simulation.

1. Biological Background

All animals are faced with the computationally intense task of continuously acquiring and analyzing sensory data from their environment. To ensure maximally useful data, animals appear to use a variety of motor strategies or behaviors to optimally position their sensory apparatus. In all higher animals, neural structures which process both sensory and motor information are likely to exist which car, coordinate this exploratory behavior for the sake of sensory acquisition. We believe the cerebellum may be involved in this motor-sensory loop.

To study this possibility, we have chosen the weakly electric fish, which use a unique electrically based means of exploring their environment^{1,2}. These nocturnal fish, found in murky waters of the Congo and Amazon, have developed electrosensory systems to allow them to detect Gojects without relying on vision. In fact, in some species this electric sense appears to be their primary sensory modality.

This sensory system relies on an electric organ which generates a weak electric field surrounding the fish's body that in turn is detected by specialized electroreceptor cells in the fish's skin. The presence of animate or inanimate objects in the local environment causes distortions of this electric field, which are interpreted by the fish. In some species of weakly electric fish, the electric organ fires a short pulse and then is silent, in effect gating the electrosensory information into the nervous system at discrete times rather than entering as a continuous stream like most other sensory modalities. Other species sample their environment with a pulse in the frequency domain, ie. by generating a nearly sinusoidal electrical discharge. The simplicity of the sensory signal, in addition to the distributed external representation of the detecting apparatus, makes the weakly electric fish an excellent animal with which to study the involvement in sensory discrimination of the motor system in general and body position in particular.

It is of value experimentally and also interesting to note that some of these fish have the largest cerebellum, relative to their brain and body mass, of any class of animals. The experiments we have undertaken are specifically aimed at understanding to what extent the exploratory behavior of the fish involves coordinated positioning of both its electric organ and its electroreceptors to resolve objects in its local electric field.

Simulations in two dimensions^{3,4} and our measurements with actual fish have shown that body position, especially the tail angle, significantly alter the fields near the fish's skin. We are currently developing freeze-frame

video techniques to be used in combination with high resolution electrode arrays positioned in the fish tank to record fish behavior in response to a variety of environmental stimuli.

To study quantitatively how the fish's behavior affects the "electric images" of objects, we are developing three-dimensional computer simulations of the electric fields that the fish generate and detect. These simulations, when calibrated with the measured fields, should allow us to identify and focus on behaviors that are most relevant to the fish's sensory acquisition tasks, and to predict the electrical consequences of the behavior of the fish with higher spatial resolution than possible in the tank.

Being able to visualize the electric fields, in false color on a simulated fish's body as it swims, may provide a new level of intuition into how these curious animals sense and respond to their world.

In this paper, we discuss a physical model of an electric fish, then the equivalent mathematical problem, which is a solution of Laplace's equation in the region exterior to the fish and the object it is sensing. We give a brief description of the Boundary Element method for solving this problem, and explain why this method is well suited for a distributed parallel architecture. Finally we describe some early results from the simulation.

2. Physical Model

We need to reduce the great complexity of a biological organism to a manageable physical model. The ingredients of this model are the fish body, shown in Figure 1, the object that the fish is sensing, and the water exterior to both the fish and the object.

The real fish has some projecting fins, and our first approximation is to neglect these because their electrical properties are essentially the same as those of water.

Our second approximation is to simplify the time-dependence of the electric field set up by the fish. The time constant associated with electric field variations in a dielectric medium is of order dielectric constant divided by conductivity⁵. For water this charactaristic time is measured in fractions of a microsecond, and for a perfectly conducting object is zero. The time between pulses of the electric organ is about a millisecond in A. leptorhynchus, so that if the fish is sensing a perfectly conducting object, it is safe to ignore time variation and model the fields as static. For some plant materials, however, this time constant may be large, and the fish may sense phase information (analogously to humans using the phase difference between the ears to sense the

direction of a sound).

In this paper, we shall concentrate on the statis approximation. There is thus an electric field, maintained by the fish, which causes a current flow proportional to the electric field according to Ohm's law.

We will assume that the fish is exploring a small conductive object, such as a small metal sphere. First we reduce the geometrical aspect of the object to being pointlike, yet retaining some relevant electrical properties. Except when the object is another electric fish, we expect the object to have no active electrical properties, but only to be an *induced dipole*, so that in the presence of an electric field the object becomes a dipole of strength proportional to the field and oriented opposite to the field. The proportionality constant is the *polarizability* of the object.

Thus the polarizability is the only parameter describing the object. In this first paper, we shall not attempt to calibrate experimental measurements and computed results, but merely estimate this parameter. Polarizability has the dimensions of volume, so we shall model an object of polarizability I cm³, since this is the size of object used in the experiments.

We now come to the modelling of the fish body itself. This consists of a skin with electroreceptor cells which can detect potential difference, and a rather complex internal structure. We shall assume that the source voltage is maintained at the interface between the internal structure and the skin, so that we need not be

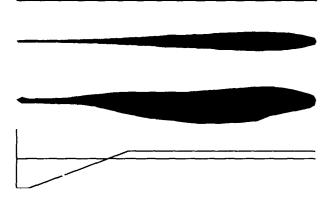


Figure 1. Top, top view of the fish Apteronotus leptorhynchus, Middle, side view of the fish. The fish is about 20 cm long.Bottom, modelled voltage profile ϕ along the interior of the fish, from -100 mV at the tail with a linear ramp to +25 mV at the head. The fins and tail are not shown.

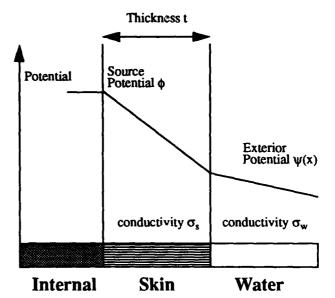


Figure 2: A section through the fish skin, with electrical potential plotted vertically. The potential is assumed linear within the skin.

concerned with the details of the internal structure. Thus the fish body is modelled as two parts: an internal part with a given voltage distribution on its surface, surrounded by a skin with variable conductivity.

Because of the voltage on the internal body, a current distribution is set up in the fish skin and water, which have different conductivities. The signal from the electroreceptor cells in the skin is assumed to depend on the potential difference across the skin⁶.

We shall simplify the model a little more by assuming that the skin thickness is small compared to the size of the fish. This is not equivalent to neglecting the skin altogether, since it is the combination of skin thickness and conductivity which determines its electrical properties; the zero-skin-thickness approximation merely removes geometrical complexity from the model in exchange for a slightly more complex boundary condition at the surface of the fish body, as discussed below.

Figure 2 shows a section through the body of the fish, with a graph of the voltage or potential superimposed. We define ϕ to be the potential at the interface between the fish skin and the internal part of the fish, and the scalar field $\psi(x)$ to be the potential field in the water exterior to the skin. The conductivities of the skin and water are written σ_{ϕ} and σ_{ψ} respectively.

We write the normal derivative of the exterior potential as ψ_n , and conservation of current then implies that the

slope of the potential in the skin be $\psi_n \ \sigma_w / \sigma_s$. We shall now assume that the potential varies linearly from the inside to the outside of the skin; sufficient justification for this would be that either the skin is thin compared to the body thickness, or that the source potential varies slowly over the skin compared to the skin thickness.

Using the thickness t of the skin, we find the boundary condition

$$\psi - \xi \psi_n = \phi \tag{1}$$

where the effective skin thickness \xi is defined to be

$$\xi = t \frac{\sigma_w}{\sigma_s}$$

This is a Cauchy or mixed boundary condition for the exterior potential.

Conservation of charge is again the guiding physical law to obtain the differential equation satisfied by ψ in the water. Mathematically, it means that the divergence of the current density is zero; thus we can use Ohm's law to write

$$\nabla \cdot (\sigma \nabla \psi) = 0$$

where σ is the conductivity of the water, assumed uniform, and $\nabla \psi$ is the electric field. This equation reduces to $\nabla^2 \psi = \uparrow$, Laplace's equation.

3. Mathematical Theory

The Boundary Element method^{7,8} has been used for many applications where it is necessary to solve a linear elliptic partial differential equation. The derivation is particularly simple for the case of Laplace's equation, which we present with less than complete mathematical rigor.

Green's theorem states that if functions U and V are free of singularities in a domain Ω , with the normal outward from Ω , then

$$\int_{\Omega} (U\nabla^2 V - V\nabla^2 U) d^3 q = \int_{\partial\Omega} \left(U \frac{\partial V}{\partial n} - V \frac{\partial U}{\partial n} \right) d^2 q$$

We define V to be the desired solution ψ , and for some fixed point p, we set

$$U(q) = \frac{1}{|p-q|}$$

Since $\nabla^2 \psi = 0$ and $\nabla^2 U = -4\pi \delta(p - q)$, Green's theorem becomes

$$\Psi(p) A^{\Omega}(p) = \int_{\partial \Omega} \left(\frac{1}{|p-q|} \frac{\partial}{\partial n_q} \Psi - \Psi \frac{\partial}{\partial n_q} \frac{1}{|p-q|} \right)$$

where $A^{\Omega}(p)$ is the solid angle around p subtended by Ω ; for example if Ω is a cube, then A is 4π inside the cube, 2π on a face, π on an edge, and $\pi/2$ at a corner of the cube.

We can simplify the notation by introducing linear operators B^{Ω} and C^{Ω} , which can be defined by their actions on a dummy function u:

$$(B^{\Omega}u)(x) = \int_{\partial\Omega} u(x') \frac{\partial}{\partial n'} \frac{1}{|x-x'|} dx'$$

$$(C^{\Omega}u)(x) = -\int_{\partial\Omega}u(x')\frac{1}{|x-x'|}dx'$$

so that the Boundary Element Theorem for Laplace's equation becomes

$$A^{\Omega}\psi + B^{\Omega}\psi + C^{\Omega}\psi_{-} = 0 \tag{2}$$

Notice that if $\Omega' = \Re^3 \backslash \Omega$, which is the region outside Ω , then $A^{\Omega} + A^{\Omega'} = 4\pi$, $B^{\Omega} + B^{\Omega'} = 0$, and $C^{\Omega} = C^{\Omega'}$.

When the function ψ is approximated with Finite Elements as discussed below, the operators A, B and C become matrices, with A diagonal.

The Boundary Element theorem (2) provides a relation between ψ and its normal derivative at any point on the surface of Ω , so that given another relation between the two (the boundary condition (1)), we can solve for both. We wish to solve for the normal derivative of the potential, so we combine the Boundary Element theorem and the boundary condition to obtain

$$(\xi A + \xi B + C) \psi_n = -(A + B) \phi \tag{3}$$

Note that this result is only true if the domain Ω is free of singularities.

In the case of our model of the fish, the domain of interest is that outside the fish and the object, extending to infinity. We have solved for the normal derivative because it is this that determines the potential difference across the fish skin, which in turn determines the response of the electroreceptor cells.

The solution of (3) yields the potential derivative for the fish with no object in its environment. The solution for the fish with object is obtained by introducing an induced dipole. Let Ψ be the potential in the presence of the dipole. Without loss of generality, we may assume the dipole to be at the origin, so that the vector strength \mathbf{d} of

the dipole is (proportional to) the gradient of ψ at the origin. This gradient may be written as a surface integral by differentiating the Boundary Element theorem:

$$\mathbf{d} \propto \nabla \psi (0) = -\int_{\partial \Omega} \left(\frac{\mathbf{q}}{q^3} \frac{\partial}{\partial n_q} \psi - \psi \frac{\partial}{\partial n_q} \frac{\mathbf{q}}{q^3} \right) d^2 q \qquad (4)$$

We now separate out the singular part of Ψ , defining Ψ_1 by subtracting the dipole contribution:

$$\Psi_1 = \Psi - \frac{\mathbf{d} \cdot \mathbf{r}}{r^3}$$

Given that Ψ_1 satisfies the Boundary Element equation (2), because it is free of singularities, and Ψ satisfies the boundary conditions, we may derive the equation satisfied by Ψ .

$$(\xi A + \xi B + C) \Psi_n = -(A + B) \left(\phi - \frac{\mathbf{d} \cdot \mathbf{r}}{r^3} \right) + C \frac{\partial}{\partial n} \frac{\mathbf{d} \cdot \mathbf{r}}{r^3}$$
...(5

4. Computational Method

In order to discretize the boundary element method, we have created a mesh of triangles covering the surface of the fish, as shown in Figure 3, using the Distributed Irregular Mesh Environment (DIME)⁹, a portable programming environment designed for calculations with unstructured triangular meshes on distributed memory parallel processors.

We discretize the field with linear Finite Elements:

$$\psi(x) = \sum_{v} \psi_{v} N_{v}(x)$$

where ψ_{ν} is the value of the field at the node ν and $N_{\nu}(x)$ is the piecewise linear function which is unity at the node ν and zero at every other node. The normal derivative can be similarly discretized.

As observed above, the operators B and C become matrices, and we define the matrix element $B_{\mu\nu}$ to be the value of,

$$(B^{\Omega}N_{\nu})(x_{\mu})$$

which is the operator B applied to the nodal basis function for node ν and evaluated at the position of node μ . Similarly for the operator C.

We can calculate these matrix elements either by Gaussian integration 10 on the triangles neighboring node v, or analytically. It is a useful check on the matrix

Figure 3: A typical mesh covering the surface of the fish, containing 190 rodes. The mesh is double-sheeted, for the two sides of the fish.



element calculation that as the number of Gauss points increases, the result approaches the analytic result.

To solve for the potential in the presence of the object, the procedure is then as follows. First we solve for the potential ψ on the surface of the fish in the absence of the dipole singularity using (3), then calculate the dipole strength as the gradient of this potential at the position of the object using (4). Now we solve for Ψ with this dipole, using equation (5). One way to visualize the result is to display $\Psi_n - \psi_n$, which is proportional to the voltage difference acros the skin, and thus contains all the electrosensory information regarding the object which is

accessible to the fish. Notice that it is the same matrix to be solved for both of these calculations, with different right hand sides. Thus it would be computationally efficient to decompose the matrix and back-substitute for each solve, rather than starting afresh each time.

For distributed memory parallel computation, we have a packaged LU solver¹¹ for full matrices with partial pivoting, and the solver is used in three stages as follows. First the user makes an initialization call, sending the matrix size; then an LU decomposition call, where the user passes a function pointer which will calculate any required matrix element; then a back-substitution stage, where the user passes a function pointer which will calculate an element of the right-hand-side vector.

The manipulation of the mesh is done redundantly in each processor, so that before the solve step each processor has an identical copy of the mesh, and is thus capable of calculating any of the matrix elements or right-hand-side elements. When the solver is initialized, the parallel decomposition of the matrix is dealt with by the solver; and it automatically balances the matrix element computation and solving between the processors without user input.

The solution vector is returned in a distributed form to the processors, and a simple combining operation across the parallel machine gives the complete solution to each processor. We may visualize the solution using a variety of the tools from the DIME environment.

This code is an example of distributed memory programming at its easiest and most efficient: the difficult part of the programming is the sequential part, which is setting up and manipulating the mesh over the fish skin, and the most time-consuming part of the computation is the setting up and solution of linear equations, which happens without any effort from the user. The parallel programming has been done in writing the matrix solver: when more such tools are available, parallel programming will become much easier.

Let us compare the Boundary Element method with a more conventional finite difference approach to solving elliptic problems.

To implement the finite difference method, we would first make a mesh filling the domain of the problem, that is a three dimensional mesh, then for each mesh point set up a linear equation relating its field value to that of its neighbors. We would then need to solve a set of sparse linear equations. In the case of an exterior problem such as ours, we would need to pay special attention to the far-field, making sure the mesh extends out far enough and that the proper approximation is made at this outer boundary.

With the Boundary Element method, we discretize only the surface of the domain, and again solve a set of linear equations, except that now they are no longer sparse. The far-field is no longer a problem, since this is taken care of analytically.

If it is possible to make a regular grid surrounding the domain of interest, then the Finite Difference method is probably more efficient, since multigrid methods or alternating direction methods will be faster than the solution of a full matrix. It is with complex geometries however, that the Boundary Element method can be faster and more efficient, on sequential or distributed memory machines. It is much easier to produce a mesh covering a curved two-dimensional manifold than a three-dimensional mesh filling the space exterior to the manifold. If the manifold is changing from step to step, the 2D mesh need only be distorted, whereas a 3D mesh must be completely remade, or at least strongly smoothed, to prevent tangling. If the 3D mesh is not regular, the user faces the not inconsiderable challenge of explicit load balancing and communication at the processor boundaries.

We feel that the existence of distributed matrix solving software makes the Boundary Element method preferable to conventional Finite Difference methods, since it is competitive in computation time, and much easier to program.

5. Results

Figure 4 shows four fish in various unlikely positions.

For this initial investigation we have chosen to set the effective skin thickness ξ to be 2 cm, after measurements by Scheich and Bullock ¹²; this figure has significant error, and of course the real fish has variable ξ over its body.

Figure 5 shows a side view of the fish with the free field ψ (no object) shown in gray scale, and we can see how the potential ramp at the skin-body interface has been smoothed out by the resistivity of the skin. Figure 6 shows the computed potential contours for the midplane around the fish body, showing the dipole field emanating from the electric organ in the tail.

Figure 7 shows the difference field $\Psi_n - \psi_n$ for three object positions, near the tail (left), at the center (middle) and near the head of the fish (right). In each case the object is 3cm above the midplane, and the fish is 21 cm long. It can be seen that the difference field, which is also the sensory input for the fish, is greatest when the object is close to the head. A better view of the difference voltage is shown in Figure 8, which shows the values of

the difference voltage on the midline of the fish, for various object positions. Again it may be seen that the maximum sensory input occurs when the object is close to the head of the fish, rather than the tail, from which the dipole field emanates.

References

- 1. T. H. Bullock and W. Heiligenberg, (eds), Electroreception, Wiley, New York, 1986.
- 2. H. W. Lissman, On the Function and Evolution of Electric Organs in Fish, J. Exp. Biol., 35 (1958) 156.
- 3. W. Heiligenberg, Theoretical and Experimental Approaches to Spatial Aspects of Electrolocation, J. Comp. Physiol., 103 (1975).
- M. Bacher, A New Method for the Simulation of Electric Fields, Generated by Electric Fish, and their Distortions by Objects, Biol. Cybern. 47 (1983) 51.
- J. D. Jackson, Classical Electrodynamics, Wiley, New York, 1975, p. 296.
- J. Bastian, Electrolocation, J. Comp. Physiol., 144 (1981)
- 7. T. A. Cruse and F. J. Rizzo (eds.), Boundary Integral Equation Method: Computational Applications in Applied Mechanics, ASME Proc. AMD-Vol. 11 (1975).
- 8. C. A. Brebbia et al., (eds), *Boundary Elements*, Springer-Verlag, Berlin, 1983.
- R. D. Williams, *DIME: A Users Manual*, Caltech Concurrent Computation Project Report C3P-861 (1990).
- 10. R. W. Cowper, Gaussian Quadrature Formulas for Triangles, Int. J. Numer. Methods Eng, 7 (1973) 405.
- P. G. Hipes, Comparison of LU and Gauss-Jordan System Solvers for Distributed Memory Multiprocessors, Caltech Concurrent Computation Project report C3P-652c, To Be Published in Concurrency, Practice and Experience.
- 12. H. Scheich and T. H. Bullock, The Detection of Electric Fields from Electric Organs, in Electroreceptors and Other Specialized Receptors in Lower Vertebrates, (A. Fesand, ed.), Springer-Verlag, Berlin, 1974.

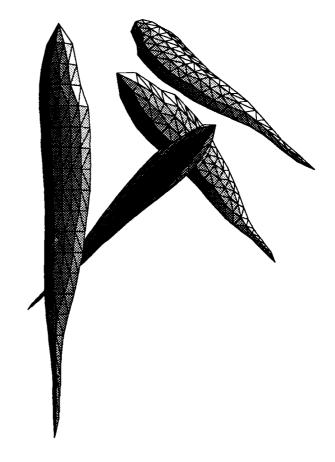


Figure 4: Four fish with simple shading.



Figure 5: Potential distribution on the surface of the fish, with no external object.

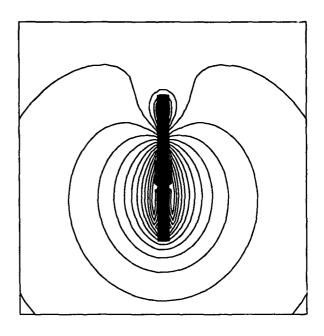


Figure 6: Potential contours on the midplane of the fish, showing dipole distribution from the tail.

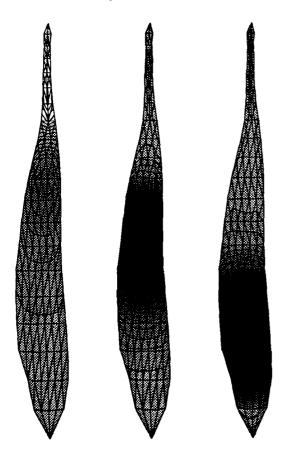


Figure 7: Gray-scale plots of voltage differences due to an object at positions (left) near tail, (middle) at center and (right) near head. Each object is 3cm above midplane.

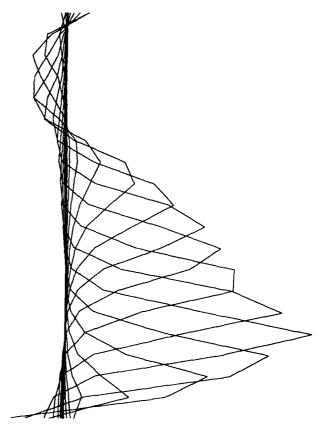


Figure 8: Envelope of voltage differences along midline of the fish, for 20 object postions, each 3cm above midplane.

Molecular Dynamics Simulations of Short-Range Force Systems on 1024-Node Hypercubes *

Steven J. Plimpton

Sandia National Laboratories Albuquerque, NM 87185

Abstract

Two parallel algorithms for classical molecular dynamics are presented. The first assigns each processor to a subset of particles; the second assigns each to a fixed region of 3d space. The algorithms are implemented on 1024-node hypercubes for problems characterized by short-range forces, diffusion (so that each particle's neighbors change in time), and problem size ranging from 250 to 10000 particles. Timings for the algorithms on the 1024-node NCUBE/ten and the newer NCUBE 2 hypercubes are given. The latter is found to be competitive with a CRAY-XMP, running an optimized serial algorithm. For smaller problems the NCUBE 2 and CRAY-XMP are roughly the same; for larger ones the NCUBE 2 (with 1024 nodes) is up to twice as fast. Parallel efficiencies of the algorithms and communication parameters for the two hypercubes are also examined.

Introduction

Molecular dynamics (MD) simulations are commonly used to calculate static (thermodynamic) and dynamic (transport) properties of liquid and solid state systems. Each of the N atoms (or molecules) is treated as a point mass and Newton's equations of motion then integrated to move each atom forward in time. The physics of the model is encompassed in the potential energy functional for the system from which individual force equations for each atom can be derived.

We are interested in a general class of MD problem that has three salient characteristics. The first is short-range forces, meaning that each atom interacts only with other atoms that are less than a cutoff distance r_c away. Many solid and liquid materials are modeled this way due to electronic screening effects. Hence the computation required is only O(N) instead of $O(N\log_2 N)$ as in the long-range force case

The second characteristic is that atoms diffuse. Thus, each atom's neighbors change as the simulation progresses. While the algorithms we develop are relevant to the fixed lattice case (neighbors of an atom remain the same throughout the simulation), it is a harder problem to efficiently maintain a list of neighbors. Any liquid simulation and most solid simulations where structure is changing require this.

The third characteristic is problem size. We consider problems ranging from a few hundred atoms to several thousand. The vast majority of work in the field is on systems of this size and many macroscopic features can be accurately modeled by such systems [1,2]. A model is typically designed with N as small as possible to capture the desired macroscopic effects. The goal is then to perform each timestep as quickly as possible since each step represents only $\sim 10^{-15}$ seconds of "real" time. In practice tens or hundreds of thousands of timesteps are needed. Thus it is more interesting to be able to do 100,000 timesteps of a 100000 atom system than 10000 timesteps of a 1000000 atom system.

As has been extensively discussed, MD algorithms are inherently parallel [3,4]. Previous work on hypercubes has demonstrated their potential for MD, but has typically been done with relatively few processors [5,6]. Our goal in this research was to implement the fastest parallel algorithm possible for this class of problem to see if it could perform as well as the best serial algorithm on a CRAY-XMP vector supercomputer. This is a difficult task, since MD algorithms can be vectorized and execute at tens of thousands of timesteps per hour on a CRAY. As we shall see, achieving these speeds with current generation hypercubes requires at least 512 processors.

In the next section the model problem is described. Then two serial algorithms are discussed along with their corresponding parallel implementa-

^{*}This work was performed at Sandia National Laboratories which is operated for the U.S. Department of Energy under contract number DE-AC04-76DP00789.

tions. Timings of the serial versions on a CRAY-XMP and of the parallel algorithms on the hypercubes are given. Finally, some comments are made with regard to comparing the two architectures and conclusions drawn as to the fastest algorithms.

Model problem

The physical system modeled is a block of Al atoms, periodic in all 3 dimensions so as to simulate homogeneous bulk material. The block sides are multiples of the Al lattice constant of 4.04Å. Atoms can be removed from the lattice to study point defects, or arranged differently and given suitable boundary conditions to study planar defects. We model it at a temperature slightly less than the melting point of Al $(930^{\circ}K)$ at constant N, V (volume), and E (energy). The interaction between atoms separated by a distance r is assumed to be pairwise and given by the Morse potential

$$\phi(r) = \phi_0 \left(e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)} \right)$$
 (1)

with constants ϕ_0 , α , and r_0 defined for Al. The potential function is then cut and shifted so as to go to zero at a distance $r = r_c$. The computational task at each time step is to integrate the set of N coupled ODE's given by

$$m\frac{d^2\vec{x}_i}{dt^2} = -\sum_{i \neq i} \frac{d\phi(r_{ij})}{dr} \frac{\vec{x}_i - \vec{x}_j}{r_{ij}}$$
(2)

where the summation is over all atoms within a distance r_c of atom i. The initial conditions are specified by choosing a system energy and corresponding initial velocities for the atoms. As the integration proceeds, the system equilibrates and various effects can be studied and calculated such as diffusion, melting, etc.

All of the algorithms discussed take advantage of several computational tricks common to MD (see [7] for example). First and foremost, the force (derivative of equation (1)) is tabulated for 10000 distances at the beginning of the simulation. When it is needed in a force calculation, a value is simply linearly interpolated from the table. This is a fast operation on the CRAY-XMP because of its gather-scatter hardware and is a significant savings because energy functionals more complicated than equation (1) (such as pseudopotentials) need only be calculated once and hence are no more costly to use. The hypercubes have ample memory for each processor to store the full table. Additionally, square roots and excess flops are avoided by calculating the forces

in r^2 space and storing velocity and force in units of distance. The simplest scheme for integrating equation (2) is also used, a leapfrog algorithm, since high accuracy in the integration is not a concern (due to the approximations inherent in the potential function ϕ).

Serial Algorithms

The basic kernel of computation required to integrate equation (2) is as follows. At each timestep, each atom calculates its distance r to each of its near neighbors. If $r < r_c$ then the force due to that neighbor is calculated. This is done in turn for every atom and the summed forces used to update velocities and positions. The key to performing these calculations efficiently is to minimize the number of neighbors that must be checked for possible interactions.

The first serial algorithm (S1) uses a neighbor list to accomplish this. For each atom we create a list of all its neighbors within a sphere of radius $r_s > r_c$ by calculating its distance to all the other atoms. This list is used for a few timesteps to calculate all pairwise interactions; then it is rebuilt before a neighbor could have moved from a distance $r > r_s$ to $r < r_c$. Building the list requires $O(N^2)$ operations, but is amortized over several timesteps.

For N > 2000 atoms it becomes more efficient to make the neighbor list in the following way. The atoms are first sorted in one dimension (the vertical). Each atom then only need examine neighbors in the sorted list that are less than a vertical distance r_s away. Hence the entire update requires only $O(N\log_2 N)$ operations due to the sort. The sorting algorithm that appears to work best for this problem is a shell sort. It is faster than a heap or quick sort because the atom list is only partially disordered from its previous state. We note that it should be possible to construct a variant of the quick sort which would work in $O(N\log_2 k)$ time where k is the maximum distance an atom has moved in the list since the last sorting.

Algorithm S1 also takes advantage of Newton's 3rd law so that an atom only need check half its neighbors. Hence a force is calculated once for each pair of atoms, not for each atom in the pair.

The second serial algorithm (S2) is similar to S1 except in the way it calculates the neighbor list. The atoms are binned into 3d boxes with sides $s \geq r_s$ and the neighbor list for each atom constructed by checking atoms in the neighboring 26 boxes (with Newton's 3rd law, only 13 boxes). We note that S2 is a much faster technique than the related algorithm which bins the atoms at every time step into

boxes of side $s \ge r_c$ and does not use a neighbor list. This is because the cost of checking all atoms in neighboring boxes every timestep outweighs the cost of periodically constructing a neighbor list from the atoms in larger boxes.

Parallel Algorithms

The first parallel algorithm (P1) is an adaptation of S1. Each processor is assigned a set of N/N_p atoms to update for the duration of the simulation, where N_p is the number of processors (nodes). After every timestep each node broadcasts its updated atom positions to every other node. This means each node receives the current xyz positions of all N atoms, which it uses to do force calculations for the next timestep. Similar to S1, each node builds a neighbor list for its subset of atoms, so that it can efficiently calculate the required forces. As in S1, the neighbor list is rebuilt every few timesteps.

The communication portion of P1 performs the following task. Each node has a small unique piece of a large vector. We want every node to end up with a copy of the full vector. This can be done quickly using the hypercube's connectivity. Each node first exchanges information with an adjacent node in the vector; it now has a contiguous piece of the vector twice as long. It then exchanges this piece with a node two positions away; then with a node four away, etc. At the last step each node exchanges half the vector with a node $N_p/2$ positions away. Thus the global accumulation of the vector is done in d exchanges (read/write pairs) where d is the dimension of the hypercube, each exchange being done with a neighboring node (in the hypercube topology). We note this method exchanges the same amount of information as the circular ring scheme suggested for the long-range force problem [3], but requires only dmessages to be sent (and read), instead of N_p . This offers a large savings on the hypercubes where the cost of message start-up is significant. It does require each node to have sufficient memory to store the entire position vectors, but this is not a difficulty on the hypercubes for the problem sizes considered here.

The computational work required in P1 for each node is not simply that of S1 divided by N_p , because Newton's 3rd law is not implemented. To do so would require the force vectors be globally exchanged at each timestep similar to the position vectors. Since, as we shall see, communication costs are roughly 50% of the total execution time for this algorithm (on the full NCUBE/ten), it would not be

effective to halve the computation at the expense of doubling the amount of communication required.

The second parallel algorithm (P2) takes more advantage of the local nature of the force interactions, similar to S2. Each processor is assigned a fixed region of space (a small box) and updates the positions of all atoms within its box in a given timestep. We require the box side to be $s \geq r_c$ so that each node need only receive information from its neighboring 26 boxes. If $r_s > s \geq r_c$ then each node must check all the atoms in neighboring boxes at every timestep. If $s \geq r_s$ then it becomes more effective, as in S2, to construct a neighbor list, use it for a few timesteps, and rebuild the list periodically.

To insure each box can get information from its 26 3d neighbors with only nearest neighbor communication between nodes, the hypercube is mapped into a 3d mesh, Gray coded in each dimension. The required information can then be acquired by each node with only 6 exchanges. First, each node passes its atom positions to its west neighbor, then to its east. Next, it passes all of its accumulated information to the north, then to the south. Finally, the entire list of atom positions is sent to its upward neighbor, then to its downward. In addition, each node must pass along special information for atoms that left its box during the previous timestep. This includes velocities, various flags, and when a neighbor list is being kept, the neighbor list for the atom itself. Each node that receives the extra information checks to see if the atom has moved into its box. Packing this information into message buffers and reorganizing each node's list of current atoms as atoms move between boxes is all extra overhead unneeded in P1.

The computation portion of P2 is similar to S2, except that again Newton's 3rd law is not implemented, since it would require force values to be exchanged. Also, although the fraction of time spent on communication is not always as high as in P1, the efficiency of the 6-exchange mechanism described above would be partially lost if each node only needed information from 13 neighbor boxes instead of 26.

Results

Algorithms S1 and S2 were implemented on a single processor of a CRAY-XMP with special attention given to insuring the critical routines (neighbor list formation, force calculation, the integration step itself) vectorized. Algorithms P1 and P2 were implemented on both the NCUBE/ten and the

NCUBE 2 hypercubes. The NCUBE/ten at Sandia has 1024 nodes and the NCUBE 2 currently has 64 nodes; the latter will soon be upgraded to 1024 nodes (maximum configuration is 8192 nodes). Both P1 and P2 ran 3.3-3.5x faster (per node) on the NCUBE 2 than on the NCUBE/ten for all problem sizes and numbers of nodes. As will be discussed below, this behavior should hold up to 1024 nodes on the NCUBE 2. Hence, the timings given for the NCUBE 2 (for more than 64 nodes) are NCUBE/ten times divided by 3.3.

The particular choice of parameters for which timings are given is $r_c = 4.04 \text{\AA}$ (2nd nearest neighbor distance in Al) and $r_s = 5.5 \text{\AA}$, with recalculation of the neighbor list done every $T_n = 20$ timesteps. The choice of r_s and T_n is somewhat arbitrary and in fact optimal choices depend on the temperature and other parameters peculiar to a particular run of the simulation. For comparison purposes with other CRAY timings [8] we chose not to implement an automated recalculation procedure; instead we use these choices as representative values. The system sizes studied were from N = 256 atoms (4x4x4 fcc lattice) to N = 10976 (14x14x14 fcc lattice).

The timings for the algorithms are listed in Table I and displayed in Fig. 1. The data shows that for the CRAY, S2 is the fastest algorithm except for the smallest problems. As the graph shows, the work it requires is linear in N. Algorithm S1 has the $O(N \log_2 N)$ sorting dependence that begins to slow it for larger N.

The P1 times are all for 1024 processors except for N=256 and N=500 which can only use 256 nodes and N=864 which uses 512. As Fig. 1 shows P1 times increase roughly linearly in N, despite the $O(N^2)$ cost of creating the neighbor list. This is because while the fraction of time spent on the neighbor list calculation increases from 3% to 54% (as N increases from 256 to 10976), the fraction spent on communication actually decreases from 72% to 39%.

The P2 times reflect the fact that boxes cannot be smaller than the potential cutoff r_c , so we are restricted to using a small number of nodes for the smaller problems. The number of nodes used by P2 is shown in the N_p column. The "Box" entry is A if P2 used small boxes $(r_s > s \ge r_c)$. A B entry indicates boxes of size $s \ge r_s$ could be used and hence a neighbor list was formed and taken advantage of as discussed in the previous section. The changes in A, B, and N_p as N increases cause the kinks in the P2 curve. Nevertheless, P2 is the fastest of all the algorithms for problems large enough to use most of the available processors.

The final two lines in Table I illustrate an impor-

Table I: CPU time (in seconds) for 100 timesteps of the algorithms on various problem sizes. The serial times S1,S2 are for the CRAY-XMP; the parallel times P1,P2 are for the NCUBE 2 (inferred from NCUBE/ten times). The N_p and Box columns refer to P2 and are explained in the text, as are the two special lines at the bottom.

N	S1	S2	P1	P2	N_p	Box
256	0.65	0.80	1.09	4.16	64	A
500	1.50	1.65	1.64	10.4	64	A
864	3.05	2.75	2.23	7.55	64	В
1372	6.10	4.25	2.90	11.8	64	В
2048	11.5	6.25	3.60	4.51	512	A
2916	19.5	9.20	5.82	8.29	512	Α
4000	31.0	12.0	7.70	10.9	512	A
5324	46.6	16.2	12.0	6.98	512	В
6912	70.2	20.8	15.7	9.82	512	В
8788	100	27.1	21.9	13.1	512	В
10976	140	34.0	30.8	15.0	512	В
4096	16x8x8			4.64	1024	A
10648	22x11x11			8.52	1024	В

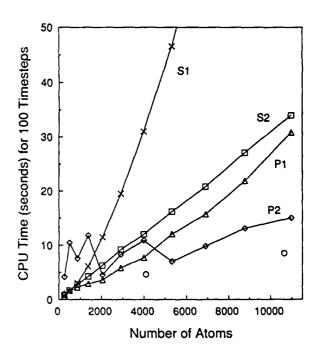


Fig. 1: Timings on the CRAY-XMP (S1,S2) and NCUBE 2 (P1,P2) for all 4 algorithms. Data values are from Table I. The isolated circles are the last two table entries. The timings indicate the NCUBE 2 (with 1024 nodes) is roughly equal to the CRAY for smaller problems and up to twice as fast on larger ones.

tant point about P2. These are two problems with dimensions somewhat artificially chosen so as to use all 1024 processors. For example, the 22x11x11 lattice (10648 atoms) is a near-optimal fit for 1024 boxes with side $s=r_s$. The fast simulation times for these two cases show the NCUBE 2 can be considerably faster than the CRAY if the physical problem size can be tailored to match the power-of-two mesh restrictions of the hypercube topology. While this is the inverse of the way the experimenter typically thinks of configuring a simulation, it is a useful trick if applicable.

Data on the parallel efficiencies of these algorithms provide a means of predicting timings for runs with larger N or with $N_p \neq 1024$. Typical timing results for increasing N_p are given in Fig. 2 for algorithm P1. The data shows that the speedup is nearly linear until $N_p = 128$. For larger N_p , the time spent exchanging atom positions (given by the dotted lines) becomes a significant factor and in fact eventually sets a limit on the speed achievable by the algorithm.

Actual timings from Fig. 2 show that for $N_p = 64$ (when there are $N/N_p = 32$ atoms per node), the efficiency is 83.3% (speed-up of 53.3). This was generally true for all problem sizes with the same N/N_p ratio. Similarly, the efficiency fell to $\sim 55\%$ on all problem sizes with $N/N_p \simeq 10$. Algorithm P2 uses only local communication, and so its efficiency is also constant for a given N/N_p ratio (although in practice, the mesh restrictions make it impossible to hold it constant as N increases). We found for boxes of size $s = r_c$ the fraction of time spent on communication was 15% and for $s = r_s$ it was 50%.

Fig. 2 also illustrates the 3.3x speed-up of the NCUBE 2 vs. the NCUBE/ten on this code for up to 64 nodes. We justify our extrapolation of this factor to 1024 nodes in the following way. For small numbers of nodes, computation is the dominating factor, and the NCUBE 2 is 3.3x faster than the NCUBE/ten (on this code) as the solid lines in Fig. 2 indicate. For larger numbers of nodes, communication effects must be considered. Communication (message passing) between neighboring nodes on the hypercubes can be modeled by the equation

$$T = \Delta_s + n\Delta_b \tag{3}$$

where T is the time for a message of n bytes to be written or read, Δ_s is a start-up time, and Δ_b is the per-byte time. The global vector accumulate discussed above can be used to determine an effective Δ_s and Δ_b where now every node in the hypercube is communicating simultaneously and so the derived Δ_s contains some synchronization and

loop overhead. Timing just the communication portion of P1 gave $\Delta_s = 500\mu s$, $\Delta_b = 1.5\mu s$ for the NCUBE/ten and $\Delta_s = 200\mu s$, $\Delta_b = 0.4\mu s$ for the NCUBE 2. Hence, there is at least the same factor of 3.3 speed-up in the Δ_b term which dominates the time required to pass the large messages used by P1 and P2. Since both the computation and communication portions of these algorithms are 3.3x faster, we expect the NCUBE 2 curves in Fig. 2 to follow the NCUBE/ten curves out to 1024 nodes.

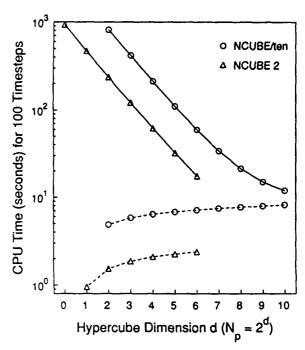


Fig. 2: Timings on the hypercubes for the N=2048 problem (using algorithm P1) as a function of nodes used. The dotted lines are the time spent in communication, which becomes a dominating factor when the full cube is used.

Caveats and Comments

- (1) The parallel results are all for single precision code. The CRAY timings are for double precision (64 bits) since that is the only option. MD codes do not typically require double precision accuracy. If it were needed, the hypercubes run in double precision (on this code) about a factor of 1.3 slower.
- (2) All the CRAY results are for one XMP processor. Some of the same techniques used in P1 and P2 could be used to adapt the serial algorithms for multiple XMP processors.
- (3) Our model problem implemented a constant NVE ensemble. Another popular choice is to hold N, P (pressure), and T (temperature) constant the

NPT ensemble. This requires rescaling the box dimensions and velocities at each timestep and would require a small additional exchange of information for algorithms P1 and P2. This communication overhead would not be present in the serial versions.

- (4) More sophisticated multi-body forces or rotational torques are often used in short-range MD simulations. This increases the amount of computation needed relative to communication. Thus, better parallel performance versus the CRAY could be expected for cases more computation intensive than our simpler pair potential example.
- (5) Though the force calculation is the key computational kernel in the MD problem, the quantities of interest are often global parameters like pressure, structure factors, diffusion coefficients, etc. These are usually calculated once every 50 or 100 timesteps and add little to the overall time required for the simulation in the serial case. The same is true for the parallel case; they can typically be calculated from each node's local information and the value accumulated quickly as a global sum.

Conclusions

In summary, we have implemented two parallel algorithms for a common MD problem, the short-range force system. Algorithm P1 exchanges global information in its communication portion, but uses only local information for computation. It has the advantage of simplicity and the ability to use more processors on small problems. Thus it is a good choice for small N. Algorithm P2 takes advantage of locality for both the communication and computation, but at the cost of significant overhead and some difficulty in mapping the physical geometry to the hypercube topology. When the 3d mesh fits well, P2 is the faster choice, particularly as N increases so that a large number of processors can be used.

On the NCUBE 2 with 1024 nodes these algorithms should be faster than vectorized CRAY-XMP algorithms for problems with more than 500 atoms. However, the difference in speed is not great except for special cases where the physical geometry maps nicely to the hypercube mesh. Nonetheless, we believe this is the first time hypercubes have been shown to be competitive with a CRAY for this class of MD problem.

While we are confident our CRAY timings are close to optimal for this problem [8], we do not claim the same for the parallel case. We expect the NCUBE 2 times to improve by a factor of two as its compilers mature and memory wait states are

reduced. Furthermore, there are two issues touched on in this work that merit further research. The first is whether a sorting enhancement to algorithm P1 (as is implemented in S1) would increase its speed.

A second issue is whether a hybrid version of P1 and P2 might be faster for some problems. For example, problems that can only use 512 3d boxes might use 2 nodes per physical box to perform the computational part without losing too much to additional communication. It is also not clear whether forcing a power-of-two mesh to fit the problem dimensions (preserving nearest neighbor communication) is always best. For example, a 10x10x10 mesh would fit some problems well and could be run on 1024 nodes, with 24 nodes idle. The issue is how to embed such a mesh into the hypercube with a minimum communication penalty. The answer might be different on the NCUBE/ten and NCUBE 2 since the latter has cut-through routing for non-nearest neighbor communication. We plan to pursue these issues in the future.

Acknowledgments

The author would like to thank Steve Cook at Cornell University for tips on vectorizing MD codes for the CRAY, particularly the use of the sorting technique. He also thanks Cindy Phillips at Sandia for useful discussions of sorting algorithms relevant to the MD problem.

References

[1] D. K. Chokappa, S. J. Cook and P. Clancy, "Nonequilibrium Simulation Method for the Study of Directed Thermal Processing", Phys. Rev. B, 39, 10075 (1989).

[2] S. J. Plimpton and E. D. Wolf, "Effect of Interatomic Potential on Simulated Grain-Boundary and Bulk Diffusion: A Molecular-Dynamics Study", Phys. Rev. B, 41, 2712 (1990).

[3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Vol. 1, (Prentice-Hall Inc., Englewood Cliffs, NJ, 1988), chaps. 9, 16.

[4] H. L. Nguyen, H. Khanmohammadbaigi, and E. Clementi, "A Parallel Molecular Dynamics Strategy", J. Comput. Chem., 6, 634 (1985).

[5] P. A. Flinn, "Molecular Dynamics Simulation on an iPSC of Defects in Crystals, in *The Third Conference on Hypercube Concurrent Computers and Applications, Vol. 2*, (ed. by G. Fox, ACM Press, 1988), p. 1303.

[6] J. B. Drake, A. K. Hudson, E. Johnson, D. W. Noid, G. A. Pfeffer, and S. Thompson, "Molecular Dynamics of a Model Polymer on a Hypercube Parallel Computer", Comput. Chem., 12, 15 (1988).

[7] M. P. Allen and D. J. Tildesley, Computer Simulation of Liquids, (Clarendon Press, Oxford, 1987).

[8] M. Schoen, "Structure of a Simple Molecular Dynamics FORTRAN Program Optimized for CRAY Vector Processing Computers", Comp. Phys. Comm., 52, 175 (1989).

Transputer Modelling of Be Star Circumstellar Discs

M. J. Coe

High Energy Astrophysics Group,
Department of Physics,
University of Southampton,
Southampton,
S09 5NH, U.K.

J. Kastner

Department of Astronomy, University of California, Los Angeles, CA 990024, USA.

Abstract

A model for the infra-red emission from the circumstellar disc of a Be star is presented. The structure and other physical parameters of this disc can be adjusted to investigate the infra-red and optical line emission from such an envelope. The model presently under investigation is based on the early work of Drake and has been computed on a μ VAXII and a Meiko Computing Surface. The parallel implementation of this model allows a more complex and realistic structure to be modelled in a reasonable timescale. Both an algorithmic and event decomposition of the code have been investigated and the two methods are compared. The model has been applied to several Be stars with good agreement with observational data.

1 Introduction

The spectra of Be stars are characterized by the presence of emission lines of hydrogen and, typically, infrared excesses. Recent studies have established a correlation between the line and infra-red intensities, and have attempted to explain this relationship in terms of models of the emission from an ionized circumstellar envelope or disc [1,2]. The optical lines and infra-red continuum are generally attributed to recombination and free-free emission, respectively. These are treated selfconsistently by Kastner and Mazzali [2] in that the same emitting region is responsible for both spectral features. The observed shape and intensity of the infra-red continuum places constraints on the density and structure of the ionized region [3], which in turn determines the emission line intensity. The predicted line strength may then be compared with observation in order to test the reliability of the model.

Here we demonstrate the improvements we are making to the approach of Kastner and Mazzali [2] through the use of the enhanced power of parallel processing. The modeling technique, discussed in Section 3, consists of combining separate computations of the line and continuum spectra of ionized slabs, each with

constant density, temperature and thickness, into an ensemble. The resulting model can therefore, in principle, describe a circumstellar disc with any desired density, temperature and/or geometric structure. In Sections 4 and 5 we describe how this modeling procedure is well suited to parallel processing, which reduces the computation time to a practical level. We also present preliminary fits to optical and infra-red observed Be star spectra obtained contemporaneously in both wavebands.

2 Transputer Systems

The transputer is the computer on a chip (processor, memory and communications) built by INMOS Ltd. It implements the communicating sequential processes (CSP) model [4] of computation of which its native language, OCCAM, is an implementation. Memory is local so the memory bandwidth grows in proportion to the number of transputers (unlike shared memory multiprocessor machines). Each transputer also has an external memory interface which extends the address space into off-chip memory, although access to this is slower than for the on-chip memory.

Transputers use point to point communication links to communicate with other transputers. Each transputer has four of these links which correspond to two OCCAM communication channels, one in each direction. These links are synchronous, bi-directional bit serial links which can sustain a data rate of up to 20Mbits/sec. They can usually be switched (either manually or electronically) so as to permit any network (subject to the restriction of four links per transputer) and as they are only point to point links, the communications bandwidth does not saturate as more transputers are used (unlike with single or multiple shared bus systems).

The T800 transputer is a 32 bit, 10 MIPS processor with 4 KByte of on-chip memory, a 64 bit floating-point co-processor (which can operate concurrently with the central processor) and capable of sustaining 1.5

Mflops. It is based on a RISC (Reduced instruction set) architecture and requires little external support logic, making it an ideal programmable building block for a concurrent system.

All the work discussed in this paper was developed on a Meiko M10 Computing Surface containing T800 transputers (the transputer array) and a T414 (the local host) which communicates via a DR11 interface to a μ VAXII computer (the host machine).

The Meiko compiler generates object code from standard FORTRAN code which can be linked into a process that is referenced from OCCAM as a library routine, communicating with its environment using channels. System channels can be used by the standard FORTRAN READ and WRITE statements to communicate with the filing system and terminal. The protocol of these channels is the same as that used by the OCCAM library routines for accessing the filing system and terminal, allowing the interconnection of both OCCAM and FORTRAN processes.

There are also user channels, which can be referenced from FORTRAN code via Meiko supplied routines. These can be used to communicate with other parallel processes. In this way FORTRAN code can be viewed as a communicating process in the same way as an OCCAM process, using the same channel protocols. Communication via these user channels is much faster than communication via the system channels due to the simpler protocol and the absence of formatting of the data.

3 A Constant Thickness Disc Model

Our computational method begins with the FORTRAN coded model developed by Drake [5,6] which uses an escape probability approach to calculate the line and continuum emission from a static hydrogen slab. The model takes into account the change in the escape probability due to the presence in the line profile of broad Stark wings. The atomic transition rate equations include all the collisional and radiative terms for energy levels up to n = 30, with angular momentum l sublevels treated explicitly for $n \le 4$. It has been shown [6] that such an approach does not cause appreciable errors for the density range considered. The slab model has uniform thickness, electron temperature and density, all of which are input parameters.

A set of slab models of given thickness and temperature but monotonically decreasing density may be combined to create a simple model of a cylindrical circumstellar disc of uniform temperature [2]. The radial density structure is a step function of the form

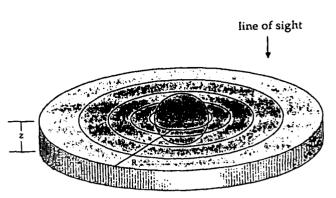


Figure 1: Schematic representation of the disc model.

 $N_{e,i} = N_{e,0}(r_i/r_0)^{-n}$ where $N_{e,i}$ is the electron density of the *i*th ring and r_i its inner radius. The overall disc spectrum is simply a sum of the individual slab models, appropriately weighted by their relative geometric areas. Thus the relevant parameters which define a constant thickness disc model are the inner disc radius, electron temperature, density index (n) and emission measure $(N_{e,0}^2 Z,$ where Z is the disc thickness). Of these, only the latter two affect the shape of the IR continuum [2].

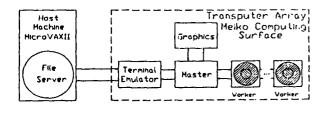
4 A Parallel Implementation of the Model

The parallel implementation of the model allows us to explore and implicitly combine the Drake models with only minor modifications to the code by using a master processor to control the overall simulation. The problem of parallel decomposition of simulations have been extensively documented [7,8]. The two approaches described here are an event decomposition and an algorithmic decomposition, which we have tried to formalize with the mathematics of orderings [9].

The changes are to some of the I/O code and provision of a harness and master process to support these changes. In this way any modifications to the Drake model do not require major changes in the transputer implementation. The input of relevant varying disc parameters is by user input channels and output of the required data by user output channels. Any non-essential screen and file output is commented out. All other file input is left unchanged, although keyboard input must be redirected to be either from a file or from a user channel. Diagnostic screen output is sent by the harness to the supervisor bus, a global low bandwidth communication bus, where it is echoed to the screen by the local host.

In this way the minimum amount of modification

is required to the original FORTRAN code. To implement an event paradigm the harness forwards the filing system using Meiko supplied routines and forwards data using a modified OCCAM load balanced pipeline [10]. Forwarding the filing system has the disadvantage that each transputer will request a copy of each file, increasing communication across the low bandwidth interface with the $\mu VAXII$ and there is also a request latency time which increases as the number of transputers in the pipe increases. However the volume of data is typically low and only loaded initially to define the model. The alternative of replacing all file input with user channel input would result in a decrease in this initial time but a large increase in the amount of modification to the FORTRAN code and to the master process.



Harness

User written Fortran

Figure 2: Decomposition of code into a pipeline.

Figure 2 shows the configuration of the system for an event decomposition of the code. The local host acts as a local file server, handling all communication with the host machine. The master processor generates the relevant data to send to each worker. Each worker processor executes its own copy of the Drake model with the parameters sent to it from the master and sends the resulting spectrum back to the master. The various slab spectra are combined by the master to produce the composite spectrum, which may be displayed on a graphics monitor as well as written to disk on the host machine.

This form of decomposition suffers from a number of problems. The first is that it requires a large amount of memory for each transputer and secondly if we are considering a single model run then there is a minimum turnaround time defined by the length of time to compute one Drake model. This is a common problem with event parallelism in that it increases throughput but does not reduce turnaround time. A further problem is that there is an overhead in emptying the pipe, when work will not be load balanced.

This is due to work being buffered further down the pipe where it is not accessible to workers earlier in the pipe. The effect of this can be reduced by limiting the degree of data buffering using buffer handshaking or alternatively by implementing a work request mechanism. Such a mechanism increases communication overheads and introduces a work request latency time, even if a more appropriate topology such as a tree is used[11].

We now consider a simple algorithmic decomposition based on a data flow network, to reduce the turnaround time and memory required by each transputer. This can be incorporated in a load balanced pipeline to further increase throughput.

The decomposition of code in to parallel communicating processes relies on identifying the data flow within the code and independent sections of computation. To identify the interdependence of a sequence of sequential processes we can define a partial order (\prec) on the code in addition to the natural total ordering (<)[12]. Consider sections of code P, Q, R then

$$P < Q, var(P) \bigcap acc(Q) \neq \{\} \Rightarrow P \prec Q$$
 and
$$P \prec Q, Q \prec R \Rightarrow P \prec R$$

where var(P) is the set of variables that P can assign to and acc(Q) is the set of variables accessed by Q that it does not initialise (cf. Bernstein's Conditions [13]). Having ordered sections of our code we have identified the interdependence since $P \not\prec Q$ and $Q \not\prec P$ implies P and Q can be performed concurrently. It also indicates how processes can be pipelined.

Clearly this ordering is only applicable to sequences of CSP processes of the form P;Q. To extend it to code embedded in control structures we can either consider only sections of code for which this is true or look at the traces (ie. the sequence of possible events) of the code. Sequences of events can be identified that are independent with the above partial order and this can be related to the FORTRAN code. The Hasse diagram of the partial order on the set of processes indicates the topology and interconnection of the processes. where each line segment symbolizes a communication link over which required shared data must be passed (cf. Precedence Graphs [14]). It also indicates any synchronization points. The direction of communication is indicated by the partial order. Such a decomposition is deadlock free due to the absence of any circular wait condition [4].

The remaining problem is to map this diagram onto the topology of the machine. This will typically involve the use of either a multipurpose harness [15] or CS tools [16]. Alternatively the processes can be mapped directly. The problems of mapping processes

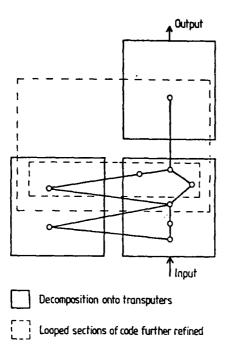


Figure 3: Hasse diagram for the Drake model

onto processors have been documented in [17].

Figure 3 shows the decomposition of the 3000 line Drake model code. The processes are mapped so as to minimise execution time and load balance computation. We have considered coarse grained decomposition of the code such that the increased communication time is less than the overlapped computation time. Further decomposition of each section of code can be performed to give a finer level of parallelisation and there will exist an order-preserving mapping between each level of refinement.

The decomposition of the code has relied heavily on the structure of the code, with most of the distributed processes being FORTRAN subroutines. In this way the process of calling and returning from a subroutine, copying the values of shared variables and results is closely matched by the effect of a communication to a CSP process [4]. The algorithmic decomposition requires less than 200K per processor, compared to 350K required for the event decomposition.

5 Results

A preliminary fit to the observed optical and near-IR continuum of a Be star, (66 Ophiuchii), is shown. The data were obtained contemporaneously in March 1988 to minimize confusion arising due to the intrinsic vari-

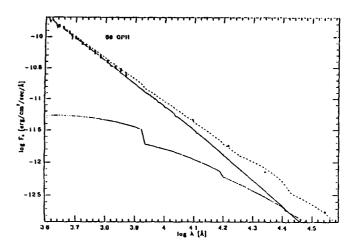


Figure 4: Preliminary fit to observed continuum of 66 Ophiuchii.

ability of Be stars. A Kurucz model stellar atmosphere [18], computed by the ATLAS6 program on a CRAY, which best corresponds to the spectral type of the central star is combined with a disc model computed as described in Section 3, to give the dashed line of total emission. A least squares fit has been achieved by normalizing the Kurucz model to the bluest optical continuum point (i.e. assuming that the stellar atmosphere dominates the spectrum at this wavelength) and then varying the disc model parameters. The fit demonstrates clearly that the combination of a stellar atmosphere with free-free emission from an ionized disc can well describe the observed continuum (as well as the Balmer emission; see [2]). However, given a comprehensive sample of Be stars, the model as presently encoded is less satisfactory in some cases (In particular, in the case of the X-ray binaries). We feel these cases may be attributed to the present state of sophistication of the model and a lack of coverage of parameter space. Both problems we expect to surmount with the use of transputers.

The time to calculate 50 constant thickness models consisting of 6 rings for 200 wavelength points is shown in Figure 5. The graph is of VAX CPU time against clapsed time on the Computing Surface, so that an observed speed up of about 50 times can be observed on a μ VAXII, when considering the event decomposition. The transputer array has the advantage that further transputers can be used with only minor modifications to the harness code to provide greater speedups for larger models. This will be required for the future enhancements to the model and to explore adequately the parameter space.

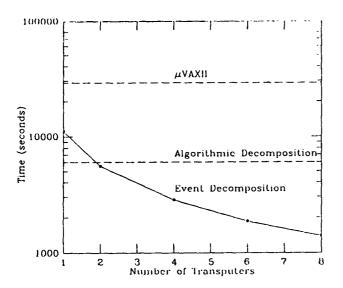


Figure 5: Timing comparison for 50 ring constant thickness model.

The efficiency [7] of the implementation is 61% for the algorithmic decomposition and > 98% for the event decomposition. This does not include the master processor, only the number of workers. For a low number of models the efficiency of the event decomposition can drop to below 90% due to work not being load balanced while the pipe is being emptied. The effect of this on our implementation has been reduced since data is not over buffered.

6 Conclusions

A substantial increase in the performance of running mathematical models can be achieved by using concurrent processors. The values given suggest 1-2 orders of magnitude reductions in run times are easily accessible when comparing the performance of a $\mu VAXII$ and an array of T800 transputers.

We note that event parallelisation is an easy form of parallelisation to implement but has disadvantages in the amount of memory required by each processor. Algorithmic decomposition requires more effort to implement and it can be difficult to obtain high efficiencies, although it requires less memory per processor and can reduce turnaround time. The same enhancements may be enjoyed by many other numerically intensive computational problems.

7 Acknowledgments

This project is primarily funded by the British National Space Centre, but support and advice from Miles Chesney and others at Meiko Ltd. as well as Steve Drake (NASA/Goddard) is also much appreciated. Martin Gorrod acknowledges an SERC studentship and the support and use of STARLINK, U.K.

References

- [1] Chokshi, A. & Cohen, M. (1987) IR Excess in Be Stars, Astron. J. 94,123.
- [2] Kastner, J. H. & Mazzali, P. A. (1989) Infrared Excess and Ho Luminosity in Be Stars: A Constant Thickness Disc Model, Astron. Astrophys. 210, 295-302.
- [3] Waters, L. B. F. M. (1986) The Density Structure of Discs Around Be Stars Derived from IRAS Observations, Astron. Astrophys. 162,121.
- [4] Hoare, C. A. R. (1985) Communicating Sequential Processes, Prentice-Hall.
- [5] Drake, S. A. (1980) The Emission Lines and Continuum from a Slab of Hydrogen at Moderate to High Electron Densities, Ph.D Thesis, University of California, Los Angeles.
- [6] Drake, S. A. & Ulrich, R. K. (1980) The Emission-Line Spectrum from a Slab of Hydrogen at Moderate to High Densities, Astrophys. J. Suppl. 42,351.
- [7] Hey, A. J. (1986) Parallel Decomposition of Large Scale Simulations in Science and Engineering, Major Developments in Parallel Processing (UNI-COM Seminar), London, December.
- [8] Fox, G. C. (1989), Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech, Concurrency Practice and Experience Vol. 1, No. 1.
- [9] Priestley, H. A. P. (1987) Lattices and Boolean Algebras, Mathematical Institute, Oxford (April).
- [10] Bowler, K. C. et al. (1988) An Introduction to OC-CAM2 Programming. Physics Department, University of Edinburgh, U.K. Chapter 5.
- [11] Pritchard, D. J. (1987) Mathematical Models of Distributed Computation, Proc. 7th OCCAM User Group Technical Meeting, Grenoble.

- [12] Lamport, L. (1978) Time, Clocks and the Ordering of Events in a Distributed System, Communications of the ACM Vol. 21, No. 7.
- [13] Bernstein, A. J. (1966) Program Analysis for Parallel Processing, *IEEE Transactions on Electronic* Computers Vol. EC-15, No. 5.
- [14] Peterson, J. & Silberschatz, A. (1985) Operating System Concepts, Addison Wesley.
- [15] Clarke, L. J. (1988) Tiny communications Harness User Manual. Edinburgh Concurrent Supercomputer Project.
- [16] Meiko Ltd. CS Tools A Technical Overview
- [17] Shen, H. (1989) Self-adjusting Mapping: A Heuristic Mapping For Parallel Programs on to Transputers Networks, Proc. 11th OCCAM User Group Technical Meeting, Edinburgh.
- [18] Kurucz, R. L. (1979) Model Atmospheres for G, F, A, B, and O Stars, Astrophys. J. Suppl. 40,1.

A Hypercube Application in Large Scale Composite Materials Modeling

C. H. Baldwin[†], S. D. Durham[‡], J. D. Lynch[‡], W. J. Padgett[‡]

† Parallel Supercomputer Initiative

† Department of Statistics
University of South Carolina
Columbia, S.C. 29208

Abstract

This large scale application combines several areas of research to develop computational models for simulating the failure mechanisms of composite materials consisting of brittle fibers (such as carbon) embedded in a matrix material (such as epoxy resin). The simulations combine the ideas of structural stress analysis, numerical linear algebra, and visualization techniques to model the behavior of fibrous composites under uniaxial tensile load. This will allow laboratory experiments to be extrapolated more accurately to real applications, providing an enhanced capability to optimize designs of large structures made of composite materials with less extensive and costly experimental programs. Further, system performance and reliability may be improved substantially. In this paper a brief discussion of the theory of composite materials as it relates to the simulations will first be given. Next the procedures used to generate and analyze the structure will be presented. The computational techniques used to perform the simulation will be given as well as results from selected test cases. A summary of results and future directions in this research will be given at the end of the paper.

Introduction

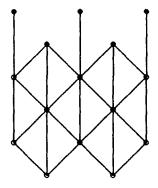
Composite materials consisting of high-strength, high-stiffness fibers embedded in a matrix are lighter than traditional materials, such as metals or wood, and are of considerable interest in current engineering practice. However, composites are not being used as much as they should be. "The basic reason surely is the uncertainly that exists in determining their strength and safe-operating lifetime in service conditions — particularly when defects could be present " [9]. It is the fibers, typically of carbon, boron, or Keylar, that give the material its uniaxial tensile strength parallel to the fiber direction. An understanding of the failure processes of such materials has been pursued for a number of years by many researchers, including Rosen [11,12], Harlow and Phoenix [7,8], Wagner, Phoenix, and Schwartz [16], Smith [13,14], Durham, Lynch, and Padgett [3,4], and others. Still a fully satisfactory theory of composite failure is yet to be achieved. It is clear that a general theory providing a bridge between standard laboratory test procedures and actual applications is highly desirable.

The structure of a single unidirectional lamina of brittle fibers in a matrix that will be utilized is based mainly on Rosen's [11] work. The unidirectional lamina to be treated consists of parallel fibers in an otherwise homogeneous matrix material. There is a bonding of the fiber surfaces with the matrix material which tends to transfer load to other fibers. Rosen [11] described the "ineffective length" δ as the length of segment around a fiber break required to redistribute the load born by the broken fiber. The composite lamina can be considered as a chain of such segments, each of length δ , referred to as the "chain of bundles" model. This important feature provides a natural discretization of the composite in the fiber direction which can be exploited computationally.

Modeling Procedures

In this application, a pinned-jointed structure depicting a unidirectional composite lamina, shown in Figure A, is utilized as a stress model.

Figure A.



The fiber centers are represented in the model as vertical line sequents, each of length δ , joined end to end while the lines of load transfer through the body of the fiber and the matrix material are represented

as diagonal line segments connected to the fiber segments at the joints. Of course, the actual fiber takes up most of the space between vertical line segments which are located at the centers of the fibers. Thus the direct forces within the fibers are experienced along the vertical segments and shear forces are transferred diagonally across the fibers, through the matrix and onto the adjacent fibers. The resulting pinned-jointed structure appears as a triangular mesh. A tensile load is applied in the fiber direction and the stresses in the members computed.

The analysis of the structure follows the mathematical methods set forth by Strang [15] for solving stress equations derived from jointed truss structures. Basically, an incidence (or in this case elongation) matrix A is formed from the geometrical nature of the structure. The matrix A relates displacements at free nodes to elongations in the attached member, while A^T relates internal forces in the members to external forces at the free nodes. In addition, a materials matrix C is formed from the elastic constants for each member. The matrix C relates elongations in the members to internal forces in the members, by the Youngs' moduli. Once these two matrices are computed, the stiffness matrix K for the structure is computed by

$$K = A^T C A \tag{1}$$

and given the displacements at the free nodes x the force balance (equilibrium equation) at the nodes f is computed by $A^TCAx = f$. Figure B outlines this procedure.

Figure B — The Stiffness Equation

Given

x: the displacements at the free nodes

Compute

Ax: the elongation of the members C(Ax): the internal forces in the members $A^T(CAx)$: the external force balance at the

nodes

The problem we wish to solve is somewhat more computationally complex, given a force balance at the free nodes f we wish to find the displacements at the free nodes x as well as the internal stresses in the members s, which amounts to computing the two matrix equations

$$(A^T C A)^{-1} f = x \qquad and \qquad C A x = s. \tag{2}$$

Note that the stiffness matrix $K = A^T C A$ is positive definite and therefore invertible. For the structure

in question, each column of A represents a particular degree of freedom for the structure while the rows with non-zero entries in that column correspond to the member attached to that node. With a regular ordering of nodes and members it is possible to compute, in parallel, the individual columns of A. Note that the maximum number of nonzero entries in any column is 6, substantially reducing the memory requirements for the column data. Likewise, with a regular ordering of members, the individual elements of C can be easily computed. The storage structure we chose to use for each column is given in Figure C.

Figure C - Column Storage Structure

column_vector:

rows = number of nonzero values in column index = integer values for non-zero member numbers value = values of non-zero entries

Thus, each column will occupy no more than 52 bytes.

The solution of the equation $(A^TCA)^{-1}f = x$ is the most computationally intensive portion of equation 2 and requires the most efficient parallel implementation. Since A and C are sparse it would be advantageous to use solution techniques which preserve their matrix structure. As the direct factorization of the stiffness matrix would necessitate computing $K = A^T C A$ and then factoring K by some method thereby losing the previously described sparsity structure, we chose to use an iterative technique and work directly with A and C. Iterative techniques in general have been shown to be very efficient when applied to sparse (as well as full) matrices and implemented on hypercube multiprocessors by Fox, et. al. [5] and Baldwin [1]. Several iterative techniques were tried in order to solve equation 2. The first method we attempted to use was the Gauss-Jacobi technique which is easily adapted to the hypercube and shows good convergence properties for a large collection of positive definite matrices - unfortunately the stress matrix could not be shown to always converge with the assumptions of this method. Given the generated matrices A and C, using the Gauss-Seidel technique we could not efficiently implement the back substitution phase of the algorithm.

Next the conjugate gradient algorithm was applied to the problem of solving equation 2. The conjugate gradient technique is described in Fox et. al. [5] and has the property that it converges for all positive definite matrices and it is easy to implement on sparse matrices. The given composition of the A and C matrices made the application of the conjugate gradient

to the problem very efficient. A number of preconditioning techniques, such as Jacobi (main diagonal) preconditioning as described in [6], can be applied to the conjugate gradient algorithm to speed convergence of the iterates but none have as yet been used in this application. The basic conjugate gradient method used to solve Ax = b is given in Figure D.

Figure D - The Conjugate Gradient Algorithm

```
x^{(0)} = \text{initial guess ( usually zero )}
r^{(0)} = p^{(0)} = b - Ax^{(0)}
k = 0
\text{not-converged} = \text{convergencetest(} r^{(0)} \text{)}
\text{while not-converged}
q^{(k)} = Ap^{(k)}
\alpha_k = \frac{\langle r^{(k)}, r^{(k)} \rangle}{\langle r^{(k)}, q^{(k)} \rangle}
x^{(k+1)} = x^{(k)} + \alpha_k q^{(k)}
r^{(k+1)} = r^{(k)} - \alpha_k p^{(k)}
\beta_k = \frac{\langle r^{(k+1)}, r^{(k+1)} \rangle}{\langle r^{(k)}, r^{(k)} \rangle}
p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}
k = k + 1
\text{not-converged} = \text{convergencetest(} r^{(k+1)} \text{)}
```

Note that there is only one matrix vector product in the conjugate gradient algorithm, which makes it advantageous for use with matrices which are sparse.

Once the stresses in the members are computed they are to compared with the strengths of that fiber component obtained from the brittle fracture [4] or the well known Weibull distribution for possible breakage. The strengths of the fibers are computed using uniform random numbers generated from the parallel linear congruential method as described in Fox, et. al. [5]. The basic sequential linear congruential algorithm used is given in Figure E.

Figure E — The Linear Congruential Method

```
m= a modulus m \geq 0

a= a multiplier 0 \leq a < m

c= an increment 0 \leq c < m

X_0= an initial value 0 \leq X_0 < m

while n \leq number_of_randoms

X_{n+1}=(aX_n+c) \bmod m
```

The choice of the values m, a, and c will greatly affect the randomness of the above algorithm, see Knuth [9]. It may also be necessary, depending upon the desired randomness of the numbers, to add a shuffling algorithm such as the on described by Bays and Durham [2].

The above techniques define the main computational techniques used in this simulation, which can be considered standard. In addition, two critical portions of the simulation are balancing the computational load between the processors and visualization of the stresses within the structure. For this particular application both of these tasks were found to have straight-forward implementations on the hypercube. With regard to load balancing, we wish to evenly distribute the work involved in solving the system. We first number the joints as well as the members starting at the upper left hand corner of Figure A and continuing vertically down then horizontally to the right and use the standard 2 dimensional coordinate axis centered on the upper lest joint. In addition each joint has two degrees of freedom, one in the x direction and one in the y direction. This gives us a regular ordering for the starting configuration of the pinned-joints and the members. It follows that there is a mathematical relationship between the indices of one joint and its neighbors, as well as the index of a member connecting two joints. Hence each processor computes, independently, a portion of the incidence, as well as a portion of the materials matrix needed to perform the matrix-vector product of the conjugate gradient algorithm. We only require, for reasons to be explained shortly, that both degrees of freedom for one joint be assigned to one processor. Thus each processor will have the same number of columns from the incidence matrix ± 2 . As the diagonal members represent lines of force transfer we may not need to explicitly include them in the display of the specimen, rather only the vertical fiber members are displayed after the stress computation. This serves to reduce the computations needed for display of the stresses. Since we have assigned both degrees of freedom for one joint to a processor we have also uniquely determined the processor which should draw the vertical member above the joint. By implementing these techniques we have eliminated the need for communication between processors in both the matrix generation and the display portions of the program.

As an additional note on reducing communication overhead, the above ideas can be used to optimize communications in the matrix vector multiply operations. The matrix-vector product Ax from Figure B results in a vector with a length of exactly the number of members in the structure. Within each processor the individual multiplications result in vectors whose non-zero components represent the contribution to total member elongation obtained from the deflection of joints assigned to that particular processor. The total elongation in all members is then the sum of the

individual processors elongations. However, depending upon the size of the hypercube, as well as the size of the problem, not every node will necessarily be assigned joints whose deflection directly affects all members in the structure. Thus, every processor can calculate the range of members directly affected by assigned joint deflections. The processors can then compute, with no communication, those processors which act upon members within its range. This implies that even though the columns are mapped into a ring of processors, it may not be necessary to shift each column through all processors. In fact, as long as more than one fiber is assigned to each processor, only one shift in each direction of the ring is required to perform the multiplication.

The above is a description of the basic procedures which are performed for a force balance applied to the structure. Once the displacements and internal stress in each member are computed, the incidence matrix should be updated before another force balance is applied. Also, when a fiber breaks, the incidence matrix should be updated to reflect one less member in the structure. In this fashion the mechanics of the structure can be visualized at each stage until complete failure is reached.

Computational Techniques

The methods described above were conceived with the idea of keeping message traffic at a minimum. To this end, collective hypercube communications where all processors participate, such as combining partial sums in an inner product or broadcasting common data, are implemented using cube geodesics as described in Fox, et al. [5] and Gustafson, et. al. [6]. Thus collective routines are O(log(P)), where P is the number of processors. If the problem size is such that one processor has the joints of more than one fiber then the communications cost in the matrixvector multiply is O(1) per iteration. This appears optimal for this application. The primary area of concern was that of space in the node processors, which is limited to 512K — of which approximately 48K is used for message buffers, 8K for the node operating system, and 4K for a jump table for operating system traps. In addition, the graphics libraries expand the executable by approximately 50K, which is approximately 20K, this leaves a grand total of about 380K for data. Also, the graphical display has a maximum size of 1024 by 768 pixels, so that one can not use all 1024 nodes and achieve maximum efficiency in the matrix-vector product as described above. Alternatives will be discussed in the closing remarks.

The following algorithms outline the code for the host and node programs in the current modeling envi-

ronment.

Host Algorithm for Composite Modeling

- H1 start host timer
- H2 get composite data; size of specimen, and Youngs' moduli
- H3 load program on nodes
- H4 send composite data to nodes via broadcast
- H5 send forcing data to nodes
- H6 wait for timing information from nodes
- H7 stop host timer
- H8 display timing data for host and node

Node Algorithm for Composite Modeling

- N1 initialize timers
- N2 initialize graphic display and load color table
- N3 receive composite data size of specimen, and Youngs' moduli
- N4 perform connection mapping for 1-D ring
- N5 construct elongation and materials matrix, C and A and find maximum and minimum number of processors to shift partial results of matrix-vector product thru
- N6 get forcing data from host, f
- N7 perform conjugate gradient to solve $A^T C A x = f$ for displacements x
- N8 perform matrix-vector multiply CAx = s to find internal stress s
- N9 perform plot of stress data s
- N10 stop timers
- N11 send timers to host

The next algorithm details step N5 in the node algorithm above.

Matrix Generation Algorithm

- G1 Get processor number and size of cube
- G2 Compute total numer of joints, number of joints per processor, and number of processors which get an extra joint
- G3 Find tile position for this processor within a ring
- G4 Find maximum and minimum member numbers which are affected by joints in this processor
- G4 For each joint in this processor compute the entries in the incidence matrix A for both degrees of freedom, as well as the location for the member above this joint and is breaking strength
- G5 For each member this processor affects, compute the entries in the materials matrix C
- G6 Search processor on the left for processors whos joints affect the same members as this processor noting how far we have to proceed
- G7 Search processor on the right for processors whos joints affect the same members as this processor noting how far we have to proceed
- G8 Combine results of G6 and G7 above to find maximum and minimum number of shifts left and right needed to perform matrix-vector product

Once each node completes the above algorithm all static data structures are set up and all processors know the length of the pipe used in the matrix-vector product in both the left and right directions. Although not specifically cited, algorithm G above uses several utility routines extensively. These routines are given below, however, because of their intuitive nature they are not detailed.

Utility Algorithms for the Generation Algorithm

node_to_cart

maps a joint number to (x,y) coordinates node_to_neighbors

maps a joint number to all neighboring joints nodes_to_members

maps two joint numbers to a member number member_to_stress

maps a member number to its Youngs' modulus

The general conjugate gradient algorithm was presented earlier, but we have modified it so that subroutines can be called to perform the matrix vector products both within the conjugate gradient algorithm and in the main node algorithm (step N8). Note the reuse of f in order to save space.

Conjugate Gradient Algorithm

```
r = p = f
f = 0
rho = \langle r, r \rangle
iterations = 0
not-converged = convergence_test( rho )
while not-converged
       y = Ap
       y = Cy
        q = A^T u
        r = r - \alpha p
        rhoprev = rho
        rho = \langle r, r \rangle
        β=rho
rhoprev
        p = r + \beta p
        iterations = iterations + 1
        not-converged = convergence_test( rho )
```

As of this writing, we are experimenting with visualization of the stress data using several different ideas, currently we are mapping the member stresses to a 4 bit color quantity while using 4 bits of intensity to contrast the ratio of stress to strength. This generates two visual affects from the data.

Results

The following tables give the runtimes along with the speedup and efficiency of some selected specimen sizes. One should note that as processors are added the runtimes drop until a point is reached where a processor needs to communicate with more than one processor on the right and left in the matrix vector product. This phenomenon occurs at low degrees of freedom, and as more degrees of freedom are added the behavior is approximately linear with respect to time until memory space is exhausted. In the tables the runtimes are from host timing data which includes the time to load the program, send data and their associated waiting times. The table for the single node case has values extrapolated, denoted with a *, from the known data. This extrapolated value is then used in subsequent calculations for speedup and efficiency values. The extrapolating function is one of the form

$$t = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \alpha_3 e^{-x}.$$

This function contains contributions, the polynomial term, from the matrix-vector multiply as well as a damping factor, the exponential term, to take communications into account. To obtain more of a response

from the data, logarithms were first applied and the results were then converted back for listing in the tables. As a note, dof refers to the degrees of freedom of the structure, and doc refers to the dimension of the hypercube.

1 node times			
dof	runtime		
120	14		
252	23		
500	50		
1530	215		
2040	342		
5050	813*		
10000	1346*		
25100	2304*		
50200	3209*		
99448	4258*		

	2 node times			
dof	runtime	speedup	efficiency	
120	13	1.05	54	
252	18	1.28	64	
500	32	1.56	78	
1530	118	1.82	91	
2040	184	1.86	93	

	4 node times			
dof	runtime	speedup	efficiency	
120	12	1.17	29.17	
252	15	1.53	38.33	
500	22	2.27	56.82	
1530	65	3.31	82.69	
2040	98	3.49	87.24	
5050	219	3.71	92.81	

	8 node	times	
dof	runtime	speedup	efficiency
120	14	0.00	0.00
252	14	1.64	20.54
500	17	2.94	36.76
1530	40	5.38	67.19
2040	56	6.11	76.34
5050	169	4.81	60.13
10000	473	2.85	35.57

	16 nod	e times	
dof	runtime	speedup	efficiency
120	12	1.17	7.29
252	13	1.77	11.06
500	16	3.13	19.53
1530	27	7.96	49.77
2040	37	9.24	57.77
5050	94	8.65	54.06
10000	248	5.43	33.92
25100	905	2.55	15.91

	32 nod	e times	
dof	runtime	speedup	efficiency
120	13	1.08	3.37
252	14	1.64	5.13
500	15	3.33	10.42
1530	21	10.24	31.99
2040	26	13.15	41.11
5050	56	14.52	45.37
10000	136	9.90	30.93
25100	472	4.88	15.25
50200	1401	2.29	7.16

	64 nod	e times	
dof	runtime	speedup	efficiency
252	14	1.64	2.57
500	16	3.13	4.88
1530	19	11.32	17.68
2040	23	14.87	23.23
5050	38	21.39	33.43
10000	80	16.83	26.29
25100	257	8.97	14.01
50200	742	4.32	6.76
99448	1990	2.14	3.34

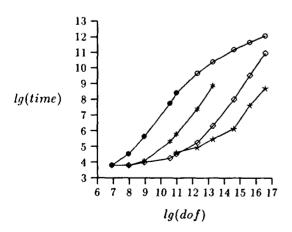
	128 nod	le times	
dof	runtime	speedup	efficiency
500	17	2.94	2.30
1530	19	11.32	8.84
2040	20	17.10	13.36
5050	31	26.23	20.49
10000	55	24.47	19.12
25100	143	16.11	12.59
50200	414	7.75	6.06
99448	1074	3.96	3.10

256 node times			
dof	runtime	speedup	efficiency
1530	20	10.75	4.20
2040	21	16.29	6.36
5050	28	29.04	11.34
10000	44	30.59	11.95
25100	92	25.04	9.78
50200	255	12.58	4.92
99448	614	6.93	2.71

512 node times			
dof	runtime	speedup	efficiency
2040	25	13.68	2.67
5050	31	26.23	5.12
10000	45	29.91	5.84
25100	71	32.45	6.34
50200	198	16.21	3.17
99448	421	10.11	1.98

The following table graphically illustrates the runtimes of several cube dimensions, namely the 0-dimensional \bullet or \circ , 3-dimensional \star , 6-dimensional \diamond , and 9-dimensional \star . The function lg(x) is the logarithm base 2. The measured values of the 0-dimensional hypercube are denoted with a \bullet while the extrapolated values are denoted with a \circ . Note that the extrapolated values for the runtimes does not seem to exactly follow the general pattern of the other runtime data; however, given the difficult task of extrapolating data — the predicted values were generated to be sufficiently conservative and most likely are greater than the extrapolated values.

Runtimes for 4 hypercube dimensions



Summary

In this application we have taken the ideas of structural stress analysis, numerical linear algebra, and visualization to produce a model for composite materials research on a hypercube multiprocessor. To date, a great deal of work has been done on providing a robust, fast code so that future enhancements can easily be incorporated. The nature of the problem lends itself well to the hypercube, and does provide many points of efficiency, in the decomposition stage for example. We believe that the code itself will help in providing new insights into the failure processes which occur in composites. The main limiting factor found in implementing these ideas is that of memory size, as this ultimately limited the size of problem we could run. Also, when dealing with this type of simulation the ability to graphically visualize the results of the modeling procedure cannot be understated. We also feel that with the new generation of hypercubes being constructed today the same ideas can be used with greater success.

Future Research

Our next idea in this area of research is to provide the ability to discretize the loading of the structure in a natural fashion, that is to iterate the algorithm outlined in the node program. This will allow us to better visualize the evolution of the structure up to complete failure. Also, the ability to display the results of computations graphically on many types of color workstations adds an incentive to view the hypercube application as a powerful "numbercruncher" and allow the data to be spooled and later displayed on a workstation more suitable for animated visualization. In the program itself the future modifications include, work on the convergence of the conjugate gradient algorithm by adding preconditioners, working with random number generating techniques, and adding the ability to discretize the loading. This last item is also of great theoretical interest as the mechanics of the structure will become very unstable as complete failure is reached.

References

- [1] Baldwin, Chuck, (1989), "Hypercube Algorithms for Successive Approximation," Proc. Fourth Conference on Hypercube Concurrent Computers and Applications, Montery, CA.
- [2] Bays, C., Durham, S., (1976), "Improving a poor random number generator,", ACM Transactions on Math Software, 2, p. 59.
- [3] Durham, S. D., Lynch, J. D., Padgett, W. J., (1988), "Inference for Strength Distribution of Brittle Fibers Under Increasing Failure Rate," J. Composite Materials, 22, p. 1131.

- [4] Durham, S. D., Lynch, J. D., Padgett, W. J., (1989), "A New Probability Distribution for the Strength of Brittle Fibers," USC Technical Report No. 138.
- [5] Fox, G., Johnson, M., Lyzenga, G., Otto, S. Salmon, J., Walker, D., (1988), Solving Problems on Concurrent Processors, Vol. 1, Prentice Hall, Englewood Cliffs, N. J.
- [6] Gustafson, J. L., Montry, G. R., Benner, R. E., (1988), "Development of Parallel Methods for a 1024-Processor Hypercube," SIAM J. Sci. Stat. Comp., 9, 609.
- [7] Harlow, D. G., Phoenix, S. L., (1978), "The Chain-of-Bundles Probability Model for the Strength of Fibrous Materials I: Analysis and Conjectures," J. Composite Materials, 12, 195.
- [8] Harlow, D. G., Phoenix, S. L., (1978), "The Chain-of-Bundles Probability Model for the Strength of Fibrous Materials II: A Numerical Study of Convergence," J. Composite Materials, 12, 314.
- [9] Kanninen, M. F., Popelar C. H., (1985), Advanced Fracture Mechanics, Oxford University Press, New York.
- [10] Knuth, D. E., (1973), The Art of Computer Programming, Vol 2: Seminumerical Algorithms. Second Edition. Addison-Wesley, Reading, Mass.
- [11] Rosen, B. W., (1964), "Tensile Failure of Fibrous Composites," AIAA J., 2, 1985.

- [12] Rosen, B. W., (1965), "Mechanics of Composite Strengthening," Fiber Composite Materials, American Society of Metals, Metal Park, OH.
- [13] Smith, R. L., (1980), "A Probability Model for Fibrous Composites with Local Load Sharing," Proc. R. Soc. Lond. A, 372, 539.
- [14] Smith, R. L., (1982), "A Note on a Probability Model for Fibrous Composites," Proc. R. Soc. Lond. A, 382, 179.
- [15] Strang, Gilbert, (1986), Introduction to Applied Mathematics, Wellesley Cambridge, Wellesley MA., 1988.
- [16] Wagner, H. D., Phoenix, S. L., Schwartz, P., (1984), "A Study of Statistical Variability in the Strength of Single Aramid Filaments," J. Composite Materials, 18, 312.

Acknowledgements

The authors would like to thank the entire staff of the University of South Carolinas' Parallel Supercomputer Initiative for their help, tolerance, and general support. The first author, Baldwin, would especially like to thank Bert Still for his technical advice, as well as TeX-nical advice as well as Terrance and Beverly Huntsberger for their help with the visualization aspects of the program. Durham, Lynch, and Padgett acknowledge support by the Army Research Office from grant DAAL-03-87-K-0101.

STUDIES OF ELECTRON-MOLECULE COLLISIONS ON THE MARK HIP HYPERCUBE

Paul Hipes, Carl Winstead, Marco Lima, and Vincent McKoy Noyes Laboratory of Chemical Physics California Institute of Technology Pasadena, California 91125

Abstract

We report on a distributed memory implementation and initial applications of a program for calculating electron-molecule collision cross sections. Runs on the Mark IIIfp hypercube show that large-grain MIMD machines are well suited for these applications. Some results of studies of $e^--\mathrm{Si}_2\mathrm{H}_6$ and $e^--\mathrm{Si}\mathrm{F}_4$ collisions will be discussed.

I. Introduction

We have developed a distributed memory implementation of a computer code which we have been using to study the collisions of low-energy electrons with molecules. Here we report on our strategy for porting this code to the JPL/Caltech Mark IIIfp hypercube, our experiences with the parallel conversion, and some initial results which illustrate the level of performance achieved. The original FORTRAN program is based on a multichannel extension of the variational principle for collisions originally introduced by Schwinger [1]. This code, which currently runs in production mode on CRAY machines, has been used extensively in recent years to study both elastic and inelastic scattering of low-energy electrons by molecules such as H₂, N₂, CO, H₂O, CH₄, C₂H₄, and C₂H₆.

Our motivations for building a hypercube version of our code for studying electron-molecule collisions include, on the one hand, the high cost of cycles on CRAY-type machines and their inherent limitations in expected CPU throughput due to the recursive character of the computationally intensive step of the calculations, and on the other hand, the potentially high performance of large-grain MIMD machines such as the NCUBE, iPSC, or the Mark IIIfp for this application, whose structure lends itself naturally to a MIMD archi-

tecture. The high-performance and cost-effective computing offered by these machines are enhancing our ability to study cross sections for collisions of electrons with industrially important gases, e.g., C_2F_6 , Si_2H_6 , and CF_3H . Such cross sections play an important role in modelling low-temperature plasmas used in plasma-assisted etching and deposition in microelectronic fabrication.

II. Background

The collision of an electron with a molecular target A may be illustrated schematically as

$$e^{-}(E_m, \vec{k}_m) + A \longrightarrow e^{-}(E_n, \vec{k}_n) + A^*$$

where the electron initially travels with kinetic energy E_m along the direction specified by the vector \vec{k}_m , and, following the collision, leaves the molecule along direction \vec{k}_n with energy E_n . The asterisk on A indicates that the molecule may be rotationally, vibrationally, or electronically excited by the collision, in which case $E_n < E_m$; collisions for which $E_n = E_m$ are referred to as elastic.

The Schwinger multichannel (SMC) procedure [2,3] is a variational method specifically formulated for obtaining the probabilities, or cross sections, for low-energy electron-molecule collision events, including elastic scattering and vibrational or electronic excitation. The SMC method is applicable to molecules of arbitrary geometry, and is capable of incorporating effects arising from polarisation of the target by the incident electron, which are particularly important at the lowest energies (approximately 0-5 eV).

In the SMC procedure, the scattering amplitude $f(\vec{k}_m, \vec{k}_n)$, whose square modulus is proportional to the cross section, is obtained in the form

$$f(\vec{k}_m, \vec{k}_n) = -\frac{1}{2\pi} \sum_{i,j} \langle S_m(\vec{k}_m) | V | \chi_i \rangle$$
$$(\mathbf{A}^{-1})_{ij} \langle \chi_j | V | S_n(\vec{k}_n) \rangle,$$

where $S_m(\vec{k}_m)$ is an (N+1)-electron interaction-free wave function of the form

$$S_m(\vec{k}_m) = \Phi_{toract}^{(m)}(1, 2, ..., N)e^{i\vec{k}_m \cdot \vec{r}_{N+1}},$$

V is the interaction potential between the electron and the molecular target, and the (N+1)-electron functions χ_i are Slater determinants which form a basis set for approximating the exact scattering wave functions $\Psi_m^{(+)}(\vec{k}_m)$ and $\Psi_n^{(-)}(\vec{k}_n)$. The $(\mathbf{A}^{-1})_{ij}$ are elements of the inverse of the matrix representation in the basis χ_i of the operator

$$A^{(+)} = \frac{1}{2}(PV + VP) - VG_P^{(+)}V - \frac{1}{N+1}\{\hat{H} - \frac{N+1}{2}(\hat{H}P + P\hat{H})\}.$$

Here P is the projector onto open (energetically accessible) electronic states,

$$P = \sum_{\ell \in open} |\bar{\Phi}_{\ell}(1, 2, \dots, N)\rangle \langle \bar{\Phi}_{\ell}(1, 2, \dots, \bar{N})|,$$

 $G_P^{(+)}$ is the (N+1)-electron Green's function projected onto open channels, and $\hat{H}=(E-H)$, where E is the total energy of the system and H is the full Hamiltonian.

In the present implementation, the Slater determinants χ_i are formed from molecular orbitals which are, in turn, combinations of Cartesian Gaussian orbitals

$$N_{\ell mn}(z - A_z)^{\ell} (y - A_y)^m (z - A_z)^n \times \exp(-\alpha |\vec{r} - \vec{A}|^2),$$

which are commonly used in molecular electronicstructure studies. With this choice, all matrix elements needed in the evaluation of $f(\vec{k}_m, \vec{k}_n)$ can be obtained analytically, except those involving the Green's-function term $VG_P^{(+)}V$. These terms are evaluated numerically via a momentum-space quadrature procedure [4]. Once all matrix elments are calculated, the final step in the calculation is solution of a system of linear equations to obtain the scattering amplitude $f(\vec{k}_m, \vec{k}_n)$.

The computationally intensive step in the above formulation is the evaluation of large numbers of so-called "primitive" two-electron integrals

$$\begin{split} \langle \alpha \beta | V | \gamma \vec{k} \rangle = \\ \int \int d^3 \vec{r}_1 d^3 \vec{r}_2 \ \alpha(\vec{r}_1) \beta(\vec{r}_1) \frac{1}{r_{12}} \gamma(\vec{r}_2) e^{i \vec{k} \cdot \vec{r}_2} \end{split}$$

for all combinations of Cartesian Gaussians α , β , and γ , and for a wide range of k in both magnitude and direction. These integrals are evaluated analytically by an intricate "black box" comprising approximately two thousand lines of FORTRAN. A typical calculation might require 109 to 1010 calls to this integral-evaluation suite, consuming roughly 80% of the total computation time. Once the primitive integrals are obtained. they are assembled in appropriate linear combinations to yield the matrix elements appearing in the expression for $f(\vec{k}_m, \vec{k}_n)$. The original CRAY code performs this procedure in two steps: first. a repeated linear transformation to integrals involving molecular orbitals, followed by a transformation from the molecular-orbital integrals to the physical matrix elements involving Slater determinants. The latter step is equivalent to an extremely sparse linear transformation whose coefficients are determined in an elaborate subroutine with a complicated logical flow.

III. Concurrent Implementation

The necessity of evaluating large numbers of "primitive" two-electron integrals makes the SMC procedure a natural candidate for parallelisation on a MIMD machine such as the Mark IIIfp hypercube. The large memory and general-purpose processors of the Mark IIIfp make it feasible to distribute the "black box" integral evaluator across the processors and to divide up the evaluation of the primitive integrals among all the processors. In planning the decomposition of the set of integrals onto the nodes of the hypercube, two principal issues must be considered. First, the number of integrals required is such that not all can be stored in memory simultaneously, and certain indices must therefore be processed sequentially. Second, the transformation from primitive integrals to physical matrix elements, which necessarily involves interprocessor communication, should be as efficient and transparent as possible. With both of these considerations in view, the approach chosen was to configure the hypercube as a logical two-torus, to which is mapped an integral matrix whose columns are labeled by Gaussian pairs (α, β) , and whose rows are labeled by momentum directions \hat{k} ; the indices $|\hat{k}|$ and γ are processed sequentially.

Given this choice of data decomposition, a design for the parallel transformation procedure must be chosen. Direct emulation of the sequential codethat is, transformation first to molecular-orbital integrals and then to physical matrix elements—is undesirable, because the latter step would entail an intricate parallel routine governing the complicated flow of a relatively limited amount of data between processors. The potential for coding errors would be unacceptably high. Instead, the two transformations are combined into a single step by using the logical outline of the original molecular-orbital-to-physical-matrix-element routine in a distributed version of the CRAY sequential routine which builds a distributed transformation matrix. The combined transformations are then accomplished by a series of large, almostfull complex-arithmetic matrix multiplications directly on the primitive-integral data set. The transformation steps and associated interprocessor communication are thus localised and "hidden" in large parallel multiplications, which are known to be efficient on hypercube architectures [13]. Besides efficiency, benefits of this approach include simplicity and enhanced portability of the resulting code.

The remainder of the parallel implementation involves relatively straightforward modifications of the sequential CRAY code, with the exception of a series of integrations over angles k arising in the evaluation of the $VG_{p}^{(+)}V$ matrix elements, and of the solution of a system of linear equations in the final phase of the calculation. The angular integration, done by Gauss-Legendre quadrature, is compactly and efficiently coded as a distributed matrix multiplication of the form $Adiag(\omega_i)A^{\dagger}$. The integration over |k| is essentially accomplished in SIMD fashion. The solution of the linear system will be performed by a distributed LU solver[14] modified for complex arithmetic, implementation of which is under way. This will make feasible solution of systems on the order of 2000 × 2000. on current hardware. However, in applications to date, the size of the linear systems—less than 100 × 100—has allowed use of the original sequential solver running either on the host or on a single node.

IV. Performance

No attempt has been made to benchmark the parallel electron scattering code in detail. Such an exercise is irrelevant here, because the integrals are embarrassingly parallel, and matrix multiplies

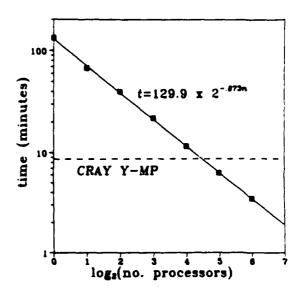


Fig. 1 Time for computation and transformation of a complete set of two-electron integrals, for fixed $|\vec{k}|$, as a function of the Mark IIIfp hypercube dimension (squares). Also shown is an exponential best fit (solid line), with parameters as indicated in the figure, and, for comparison, the single-processor CRAY Y-MP time (dashed line).

and LU decomposition have been previously assessed. However, its performance relative to the original CRAY code has been assessed through a series of calculations on Mark IIIfp hypercubes of dimensions from 0 (a single processor) to 6 (64 processors), the largest currently available. The same calculation was also performed on a CRAY Y-MP with the original code. For these comparisons, a modest but realistic "production run" for the CO molecule using 32 Cartesian Gaussian orbitals was chosen. Results are presented in Figs. 1 and 2. Figure 1 shows the time required for a single "quadrature shell" of integrals, i.e., for evaluation and transformation of a complete set of two-electron integrals for a fixed magnitude |k|, as a function of the cube dimension. All I/O and code loading are included in timings. The Weitek XL floating point processor performs the primitive integral calculation at roughly 0.85 Mflops per processor. The transformation to physical matrix elements proceeds at 1.5 Mflops/processor. The data of Fig. 1 are presented in an alternative fashion in Fig. 2, which shows speedup as a function

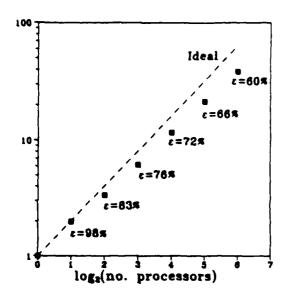


Fig. 2 Speedup as a function of hypercube dimension for the same case as Fig. 1. Efficiencies are indicated for each dimension. For comparison, ideal 2ⁿ speedup is shown by the dashed line.

of cube dimension, along with efficiencies (the ratio of achieved to ideal, or 2", speedup). The (single-processor) Y-MP time is indicated by the dashed line. As seen from the figure, the Mark IIIfp performance surpasses that achieved on the CRAY in going from 16 to 32 processors. The solid line, which is an exponential best fit, evidently describes the observed Mark IIIfp times well over the range of hypercube dimensions studied, although the fact that the time decreases as 2-0.87n rather than 2-n indicates that the speedup achieved is less than ideal. An analogous plot for the total CO computation time on 8 to 64 processors (not shown) reveals identical characteristics, reflecting the dominance of the two-electron integrals in the calculation. As expected for a problem of fixed size, the efficiency declines as the hypercube dimension increases [5], but remains reasonable over the range studied. Most importantly, on 64 processors, we are outperforming the Y-MP by a factor of 3 on a small problem. Larger problems will provide a greater performance differential.

V. Selected Results

After development and debugging, the concur-

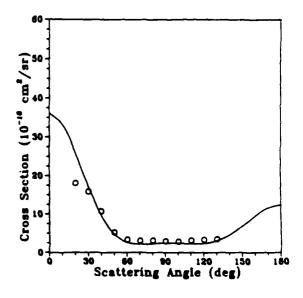


Fig. 3 Differential cross section for elastic scattering of 4eV electrons by the Si₂H₆ molecule. The solid line shows theoretical results obtained on the Mark IIIfp; the circles are measured values (Ref. [7]).

rent SMC code was applied to a number of elastic electron-scattering problems, with an emphasis on polyatomic gases of interest in low-temperature plasma applications [6]. Some of the systems examined to date are ethylene (C2H4), ethane (C2He), disilane (Si2He), and tetrafluorosilane (SiF₄). Illustrative results are presented in Figs. 3-5, along with experimental or other data for comparison [7-11]. Figure 3 shows the differential cross section—that is, scattering probability as a function of the angle & between incident and outgoing directions—for 4 eV electrons colliding elastically with Si₂H₆ molecules. Agreement with recent experimental results [7] is excellent. One point to observe is the significant probability of scattering in the high-angle, or near-backward, directions, for which experimental data are unavailable. Examination of Fig. 3 suggests that extrapolation of the measured values to this region is likely to underestimate the cross section. This fact is significant because such backscattering makes a large contribution to the transfer of momentum from the electrons to the gas molecules and is therefore important in the numerical modeling of plasmas and discharges.

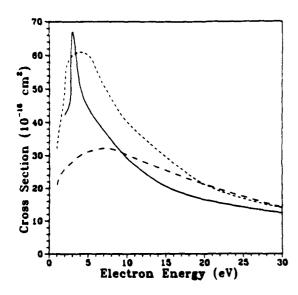


Fig. 4 Momentum-transfer or diffusion cross sections for low-energy electrons colliding with Si₂H₆. Shown are the present results (solid line), estimated values [8] (long dashes), and derived values [9] (short dashes).

The large backscattering probability indicated in Fig. 3 contributes to the peak in the Si₂H₆ momentum-transfer cross section—essentially a weighted integral over the differential cross section—shown in Fig. 4 as a function of electron energy. The dashed curves in Fig. 4, which represent estimated [8] and indirectly derived [9] momentum-transfer cross sections, appear to be the only previously published values for this industrially important molecule, highlighting the need for calculations of the present type.

As a further example of the applications performed to date, Fig. 5 shows preliminary results for the angle-integrated elastic scattering cross section of SiF₄, along with two measurements [10,11] of the total scattering cross section, which should of course be larger than the elastic cross section. Considering the uncertainties in the measurements and the need for further refinement of the theoretical result, the agreement in magnitude and overall shape of the cross sections are quite encouraging.

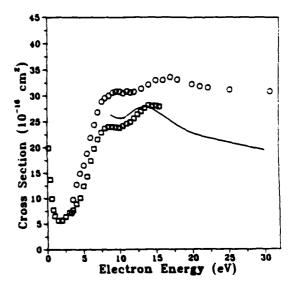


Fig. 5 Elastic electron scattering cross section for SiF₄ obtained on the Mark IIIfp (solid line). Also shown are total scattering cross section measurements of Refs. [10] (squares) and [11] (circles).

VI. Conclusions and Future Prospects

The concurrent implementation of a large sequential code which is in production on CRAYtype machines is an example of challenges which are likely to become increasingly frequent as commercial parallel machines proliferate and as more and more "mainstream" computer users are attracted by their potential. Several lessons which emerge from the port of the SMC code may prove useful to those contemplating similar projects. One is the value of focusing on the concurrent implementation of the existing code [11] and, so far as possible, maintaining the structure and code from the sequential program. The development of an understanding of the original CRAY code and its organisation is a demanding part of such a parallelisation. On the other hand, major issues of structure and organisation which bear directly on the parallel conversion deserve very careful attention. In the SMC case, the principal such issue was how to implement efficiently the transformation from primitive integrals to physical matrix elements. A poor parallelisation of the transformation could offset the high efficiency of the primitive integral calculation. The solution arrived at

not only implied that a significant departure from the sequential code was warranted but also suggested the data decomposition. One conclusion is that similar code reorganization-building and multiplying large matrices-would improve the execution on the CRAY. In contrast, the primitive integral evaluation could not be significantly improved for the CRAY because it is a recursive procedure; however, it was easily parallelised for a large grain machine. A final point worth mentioning is that the conversion was greatly facilitated by an environment which fostered collaboration between workers familiar with the original code and its application and workers adept at parallel programming practice, and in which there was ready access both to smaller machines for debugging runs and to larger, production machines.

Plans for the near future include the implementation of the distributed LU solver, already mentioned, and the implementation of portions of the sequential code necessary for studies of electronic excitation and for employing molecular symmetry to reduce computation. Subsequent steps will probably include optimisation of key sequential subroutines and transfer of the code to other parallel machines as they become available.

Acknowledgments

It is a pleasure to thank the following individuals for their encouragement and support: Don Austin of the Department of Energy; Terry Cole, Dave Curkendall, and Edith Huang of the Jet Propulsion Laboratory; and Geoffrey Fox, Paul Messina, and Heidi Lorens-Wirsba of the Caltech Concurrent Computation Program. Financial support by the Applied Mathematical Sciences Program of the Department of Energy, the Innovative Science and Technology Program of SDIO through the Army Research Office, the National Science Foundation, and the Jet Propulsion Laboratory is also gratefully acknowledged.

References

- [1] J. Schwinger, Phys. Rev. 72, 742 (1947).
- [2] K. Takatsuka and V. McKoy, Phys. Rev. A 24, 2473 (1981).
- [3] K. Takatsuka and V. McKoy, Phys. Rev. A 30, 1734 (1984).

- [4] M. A. P. Lima, L. M. Brescansin, A. J. R. da Silva, C. Winstead, and V. McKoy, Phys. Rev. A 41, 327 (1990).
- [5] G. Fox, M. Johnson, G. Lysenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, vol. I (Prentice-Hail, Englewood Cliffs, N. J., 1988), p. 58.
- [6] Plasma Reactions and Their Applications, Japan Materials Report by Japan Technical Information Service (ASM International, Metals Park, Ohio, 1988).
- [7] H. Tanaka, L. Boesten, H. Sato, M. Kimura, M. A. Dillon, and D. Spence, 42nd Annual Gaseous Electronics Conference, Palo Alto, 1989, and private communication.
- [8] M. Hayashi, Proceedings of the VI Dry Process Symposium, Tokyo, 1984.
- [9] M. Hayashi, in Swarm Studies and Inelastic Electron-Molecule Collisions, edited by L. Pitchford, V. McKoy, A. Chutjian, and S. Trajmar (Springer, New York, 1985), p. 167.
- [10] H.-X. Wan, J. H. Moore, and J. A. Tossell, J. Chem. Phys. 91, 7340 (1989).
- [11] C. Ma, P. B. Liescheski, and R. A. Bonham, XVI International Conference on the Physics of Electronic and Atomic Collisions, New York, 1989: Abstracts of Contributed Papers, p. 356, and private communication.
- [12] G. Fox, private communication.
- [13] P. Hipes, "Matrix Multiply on the JPL/Caltech Mark IIIfp Hypercube." C3P-746. G. Fox, A. J. G. Hey, and S. Otto, Parallel Computing 4, 17 (1987).
- [14] P. Hipes, "Comparison of LU and Gauss-Jordan System Solvers for Distributed Memory Multicomputers." C3P-652b.

Modeling High-Temperature Superconductors and Metallic Alloys on the Intel iPSC/860*

G. A. Geist, B. W. Peyton Mathematical Sciences Section Oak Ridge National Laboratory

W. A. Shelton, G. M. Stocks Metals & Ceramics Division Oak Ridge National Laboratory

Abstract

Oak Ridge National Laboratory has embarked on several computational Grand Challenges, which require the close cooperation of physicists, mathematicians, and computer scientists. One of these projects is the determination of the material properties of alloys from first principles and, in particular, the electronic structure of high-temperature superconductors.

The physical basis for high Tc superconductivity is not well understood. The design of materials with higher critical temperatures and the ability to carry higher current densities can be greatly facilitated by the modeling and detailed study of the electronic structure of existing superconductors.

While the present focus of the project is on superconductivity, the approach is general enough to permit study of other properties of metallic alloys such as strength and magnetic properties.

This paper describes the progress to date on this project. We include a description of a self-consistent KKR-CPA method, parallelization of the model, and the incorporation of a dynamic load balancing scheme into the algorithm. We also describe the development and performance of a consolidated KKR-CPA code capable of running on CRAYs, workstations, and several parallel computers without source code modification.

Performance of this code on the Intel iPSC/860 is also compared to a CRAY 2, CRAY YMP, and several workstations. The code runs at over 1.6 Gflops on a 128 processor iPSC/860. Finally, some density of state calculations of two perovskite superconductors are given.

1 Introduction

The discovery of high temperature superconductivity in 1986 has provided the potential of spectacularly energy-efficient power transmission technologies, ultra-sensitive instrumentation, and other devices using phenomena unique to superconductivity. Each year new materials are found to add to the family of existing high temperature superconductors. In general these materials are difficult to form and use, and some of the superconducting compounds are unstable. These difficulties are exacerbated by the lack of an accepted theory explaining superconductivity at the higher temperatures.

To further our understanding of the behavior of solids in general and superconductors in particular, the quantum mechanical laws have been formulated into sophisticated computer algorithms which can predict from first principles the structural, vibrational, and electronic properties of matter.

Present calculations of the electronic structure of real materials usually employ a mean field approximation in which each electron is viewed as moving independently in a self-consistent potential due to all of the electrons and nuclei. According to density functional theory, it is possible to express the energy of any system of electrons and nuclei as a unique functional of the electron density [1,2,3]. Since this functional is not known exactly, it is usually approximated by that appropriate to a homogeneous electron gas. This local density approximation to density functional theory has been very successful when applied to metallic and semiconducting systems, but it appears inadequate to explain important physical phenomena such as optical band gaps and superconductivity found in transition metal oxides.

More sophisticated treatments of the many electron problem are possible but have not been attempted previously because the Green's function and the sus-

^{*}This research was supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy, under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

ceptibility function that are needed to construct the electron self-energy are very difficult to calculate for real systems, especially those with narrow bands such as transition metal oxides.

Our approach is based on theoretical advances growing out of work on the Korringa, Kohn, and Rostoker coherent potential approximation (KKR-CPA) theory of alloys and magnetism [4,5,6]. The advantage in using the KKR-CPA approach is that it directly yields the Green's function for the system and thereby a direct way of calculating susceptibilities.

The effects of disorder are treated in the CPA, which is an analytic technique for calculating the configurationally averaged Green's function [7]. The KKR theory is the natural method for implementing the CPA, because it is a Green's function method and there is a natural separation between the lattice and potential.

Over the last couple of years, researchers at ORNL and their colleagues have developed a non-self-consistent semi-relativistic KKR-CPA computer code that can handle multiple atoms per unit cell. The code has wide applicability to situations in which some form of substitutional disorder plays an important role, including metallic alloys, high temperature superconducting compounds, metallic magnetism, and metal-insulator transitions.

There are three primary reasons for parallelizing this code. First, the KKR-CPA calculations are computationally intensive. It commonly requires 10 hours of CPU time on a CRAY 2 to perform a single KKR-CPA calculation. It has been estimated that over 1000 hours of CRAY CPU time would be needed to complete a single self-consistent computational experiment. The turn-around time for such experiments makes them prohibitive on existing serial computers.

Second, the KKR-CPA algorithm has a few points of natural parallelism that can be exploited to increase computational throughput. The point we will exploit is the calculation of the Density of States (DOS) at a given energy level. In order to calculate the Fermi level, it is necessary to calculate the DOS at over a hundred energy levels. Each of these DOS can be calculated independent of the other energy levels.

Third, the availability of a parallel Gflop computer, iPSC/860, has made it feasible and attractive to develop an efficient parallel version of the KKR-CPA code.

The modifications to the KKR-CPA code were made in such a way that the code could still be run on CRAYs and scientific workstations. Having only one consolidated code has made the problems of software changes and data structure interfacing much simpler than trying to keep three versions of the code up-to-

date. Moreover, the user interface is identical across all the machines the code runs on, which has been an important factor in getting scientist interested in executing this code in a parallel environment. The parallelism is hidden from the user. Even operations like getting a number of processors and loading programs onto these processors is done automatically by the code. If the user wishes to increase the parallelism, the input file contains the number of processors the computational experiment will use.

In the next section, we describe the KKR-CPA approach and how it has been parallelized for the Intel iPSC/860. In the last section, we present performance results comparing our implementation of this algorithm on several computers, and we present results from two scientific studies of the effect of alloying in perovskite superconductors.

2 KKR-CPA Algorithm

Figure 1 shows a general schematic of how we organized the consolidated KKR-CPA code. Organizing the code in this way required only a few additional routines to be written. None of the additional routines involved calculations, so exactly the same computational routines are called in the serial and parallel versions.

We opted to use a master/slave paradigm in our parallel implementation. In this scheme one processor controls work on the entire problem, and the rest of the processors perform work requested by this master process. The master process in our implementation is called the pseudo-host and executes on one of the iPSC/860 nodes. We avoided using the iPSC/860 host as the master process because of the computational imbalance between this 80386 based processor and the more powerful i860 based node. The host is also over burdened with executing the Unix operating system.

The KKR-CPA algorithm is organized in the following way. We start by inputing the atomic numbers of the species and an initial guess for the charge density and potentials.

Since the Green's function for the system at any energy is independent of any other energy, this is a natural point in the algorithm for parallelism. In the parallel implementation, the energies to be evaluated are held in a queue of tasks. The difficulty of each task is initially unknown, so a heuristic is used to order the queue in approximately decreasing difficulty. Each idle processor selects the next task in the queue and returns the density of states to the master process, which computes the integral over all energies.

This integrated density of states is used to obtain the Fermi level, which is the highest state occupied by an electron.

Load balancing is achieved naturally since all the processors will remain busy as long as there are tasks left in the queue. Each task in the queue performs the following operations.

It solves the one-electron Schrödinger equation for a single, spherically symmetric muffin-tin potential to obtain the wave functions and the scattering phase shift. The phase shift is used to construct the single site transfer matrix t, which depends only on energy and is used in setting up the KKR matrix.

The systems we are considering are periodic in space, so we work in reciprocal space by applying 3D Fourier transforms. A Wigner-Seitz cell in reciprocal space is called a Brillouin zone. Since everything is periodic in reciprocal space, we do all our CPA calculations within the first Brillouin zone.

The CPA iteration calculates the coherent single site transfer matrix t_c and the scattering path operator τ for the disordered system. Initially, t_c is approximated by the average t matrix approximation, which is the concentrated weighted average of the alloying components. The next two steps of the CPA iteration are the most computationally intensive of our approach. The processor must form the KKR matrix and then integrate its inverse over the first Brillouin zone. If there is symmetry within the Brillouin zone this can be exploited to decrease computation. For example, materials with cubic symmetry require that only 1/48 of the Brillouin zone be integrated.

To form the KKR matrix $(t_c^{-1} - G)$, it first calculates the "structure constants" matrix G. In general, the calculation of G is very difficult, but this algorithm has been made more efficient by using special polynomial fitting technique to evaluate G. A description of this method of calculating structure constants can be found in [8].

One problem in inverting the KKR matrix is it will be singular in certain regions, and it is these singularities that determine the energy bands. The KKR method can be analytically continued into the complex energy plane. By performing these calculations in the complex energy plane these singularities obtain a Lorentzian broadening and the amount of broadening is proportional to the imaginary part of the energy. In addition, due to the sensitivity of the calculation, double precision complex arithmetic is used. To evaluate the integral, hundreds or possibly thousands of complex double precision matrices of order between 80 and 300 must be formed and inverted. Each matrix corresponds to a different vertex of the tetrahedrons into which the Brillouin zone has been

subdivided. The result of the tetrahedral integration is the scattering path operator τ .

This τ is inserted into the Coherent Potential Approximation (CPA) equations to calculate the next approximation to t_c .

Once τ and t_c have converged, the Green's function for the system is calculated by combining τ and the wave function solutions to the single scatterer Schrödinger equation. The DOS for this energy is the imaginary part of the integration of the Green's function over the Wigner-Seitz cell.

Self-consistency of the charge density will soon be incorporated into the KKR-CPA code. This outer iteration involves integrating the Green's function over energy to get the charge density, which is used to obtain the potential for the next iteration. Thus the entire process described so far may be iterated several times in the self-consistent version of the code. In the parallel implementation this will involve the pseudo-host integrating the density of states it receives from the nodes over all the energies. A future paper will describe this work.

3 Results

The code has been written so that it executes on serial computers such as workstations or CRAYs as well as on parallel computers such as the Intel iPSC/860. The code requires that a minimum of 3 Mbytes of memory be available, and for the more complicated materials up to 8 Mbytes of memory is required by individual processors. Work is underway to reduce the memory requirements for the complicated materials.

The Intel iPSC/860 multiprocessor at ORNL has 128 RX nodes and 4 I/O nodes. The I/O nodes connect the RX nodes to the Concurrent File System (CFS), which has 5.2 Gbytes of disk storage. Each of the RX nodes contains a 40 Mhz Intel i860 RISC processor and 8 Mbytes of memory. The i860 has a 2 Kbyte on-chip cache and a claimed peak rate of 80 Mflops (single precision). While our hand coded assembly language BLAS routines execute on one processor at 19 – 55 Mflops, these rates are not obtained inside an application because of memory access delays. For example, in the KKR-CPA code we use the BLAS routine ZAXPY, which executes at approximately 18 Mflops inside the application.

The high Tc perovskite superconductors Ba_{1-x}K_xBiO₃ and BaPb_{1-x}Bi_xO₃ with critical temperatures of 30 K and 13 K respectively are attractive systems on which to begin a systematic study of high temperature superconductivity because their relatively simple structure (cubic) allows a more thor-

ough treatment of their electronic structure. It is believed that the superconducting state of these materials can be understood by studying their electronic structure in their normal state. Even if the mechanism for superconductivity is different in the noncuprate superconductors, because of their high transition temperature the electron-phonon coupling constant would have to be extremely large and a strong coupling of this magnitude would be a very interesting phenomenon.

The code has been run successfully on several computers using a test problem involving the high temperature superconductor (Ba.5K.5)BiO₃. The test problem required the calculation of the density of states for a fixed number of representative energies without iterating to self-consistency. The average Mflop rate for 5 computers is shown in Figure 2.

Only one processor on the CRAY 2 and CRAY YMP is used. The 130 Mflops shown in the table is achieved by modifying several routines in the basic code to further enhance vectorization. The inversion routines had already been vectorized, but the routines to form the KKR matrix had not been vectorized in the basic code.

The rate shown for the iPSC/860 includes the time to load the problem onto 128 processors, all communication, file I/O (four fairly large output files are generated), and dynamic load balancing overhead. The rate of 660 Mflops corresponds to compiled FORTRAN on a machine running at 32 Mhz. This rate was increased to 1.3 Gflops by using an assembly language BLAS routine ZAXPY in the inversion routine. When the iPSC/860 was upgraded to 40 Mhz, the ZAXPY version of the superconductor code executed at an aggregate rate of 1.6 Gflops on 128 processors.

The calculation of electronic states of alloys over a large energy spectrum is not feasible on most of the computers listed in Figure 2. But these calculations have been performed on the Intel. The first research question we asked was: What are the effects of alloying on the density of states for the two perovskite superconducting compounds $Ba_{1-x}K_xBiO_3$ and $BaPb_{1-x}Bi_xO_3$?

The rigid-band approximation has been used previously to study the effects of disorder in the perovskite superconductors [9,10]. In the rigid-band approximation it is assumed that the difference in the phase shifts of the alloying components is negligible and therefore, the effect of alloying is to rigidly shift the Fermi-energy up or down depending on whether the alloying component's atomic number is greater than or less than the original component's atomic number. The rigid-band approximation is valid only in the weak scattering limit. It is the purpose of these

calculations to test the appropriateness of this assumption.

To study the effects of disorder the density of states (DOS) of the order materials (x=0 and x=1) are calculated and compared to the disordered alloy (x=.5). At the top and bottom of Figure 3 are the DOS of the ordered compounds BaBiO3 and KBiO3 respectively and the disordered alloy Ba,5K,5BiO3 is in the middle. The DOS of these materials near the Fermi-energy $(E_t = 0.0 \text{ Ry.})$ is dominated by Bi-O states. Comparing these states we can see that the Bi-O states in the alloy have been slightly broadened by the disorder on the Ba-K sublattice. The broadening is small because Ba-K are on a different sublattice and therefore, this is a second order effect. We also found that the variation of E_f versus concentration in the CPA agrees with the rigid band approximation. Therefore, because of the small broadening of the DOS and the agreement of the variation of E, with concentration, we conclude that the use of the rigid-band approximation is valid for this material.

Similarly, Figure 4 displays the results of alloying with Lead, but here the alloying is on the Bi sublattice rather than on the Ba sublattice. At -.60 Ry and -.50 Ry in the alloy are the Bi-6s and Pb-6s states respectively and these show significant disorder. But these states are far from E_f and are unimportant. The states near E_f are Bi-O and Pb-O and these show almost no broadening. Even though the DOS show very little disorder, the variation of E_f with concentration does not satisfy the rigid-band approximation and this is the most stringent criteria that must be satisfied. Therefore, we conclude that the rigid-band approximation used previously by other authors to study the effects of alloying, is not valid for this system.

The iPSC/860 required about one hour to generate the data used in each of these figures. The results show that the superconductivity is affected in different ways by each of these alloys. The alloying with Potassium leaves the band structure essentially unchanged but decreases the Fermi energy On the other hand, alloying with Lead causes a softening of the band structure.

The use of parallel computation and the iPSC/860 has led to over an order of magnitude improvement in computational speed compared to the CRAY supercomputers for our KKR-CPA code. From a research standpoint the turnaround time for computational experiments is closer to two orders of magnitude. This greater computational power allows us to begin investigation of many unanswered questions in superconductivity and material science.

References

- [1] P. Hohenberg, W. Kohn, Inhomogeneous Electron Gas. Phys. Rev. Vol. 864, B864 (1964).
- [2] W. Kohn and L. J. Sham, Self-consistent equations including exchange and correlation effects. Phys. Rev., Vol. 140, A1133 (1965).
- [3] U. Von Barth, Density Functional Theory for Solids. In P. Phariseau and W. M. Temmerman, editors, The Electronic Structure of Complex Systems, pages 67-140, Plenum Press, New York, NY, 1984.
- [4] J. Korringa, On the calculation of the energy of a Bloch wave in metal. Physica, Vol. 13, 392 (1947).
- [5] W. Kohn, N. Rostoker, Solution of the Schrödinger equation in periodic lattices with an application to metallic lithium. Phys. Rev. Vol. 94, 1111 (1954).
- [6] G. M. Stocks, W. M. Temmerman, and B. L. Györffy, Aspects of the Numerical Solution of the KKR-CPA Equations. In P. Phariseau and B. L. Györffy and L. Scleire, editors, Electrons in disordered metals and metallic surfaces., pages 193-221, Plenum Press, New York, NY, 1979.
- [7] P. Soven, Application of the Coherent Potential Approximation to a system of muffin-tin potentials. Phys. Rev. Vol. 156, 809 (1967).
- [8] W. A. Shelton. Jr., The n-atom per unit cell KKR-CPA applied to the electronic structure of Ba_{1-x}K_xBiO₃. PhD. Thesis unpublished pages 38-57 (1989)
- [9] L. F. Mattheiss and D. R. Hamann, Electronic structure of the high Tc superconductor Ba_{1-x}K_xBiO₃. Phys. Rev. Lett., Vol. 60, 2681 (1988).
- [10] L. F. Mattheiss and D. R. Hamann, Electronic structure of BaPb_{1-x}Bi_xO₃. Phys. Rev. B, Vol. 28, 4227 (1983).

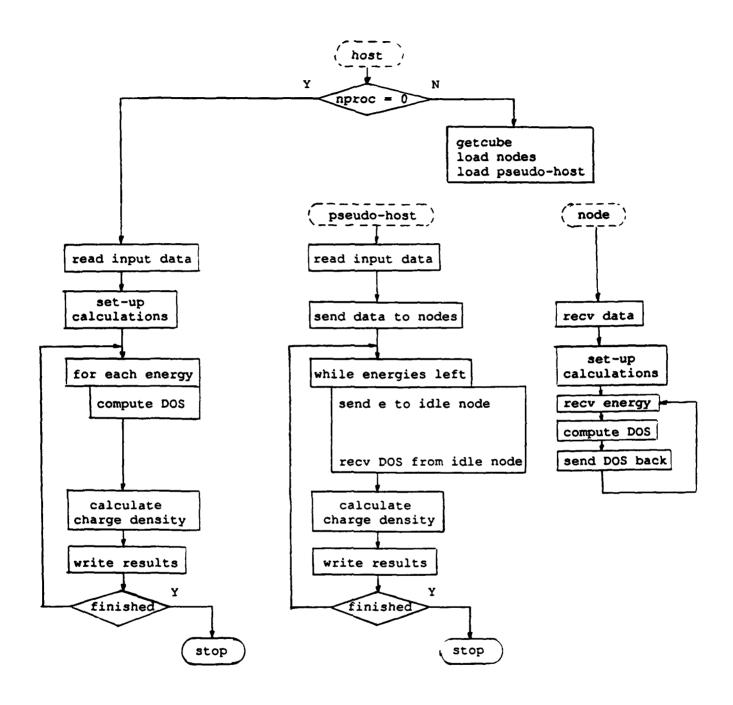


Figure 1: Schematic of parallel implementation of KKR-CPA code.

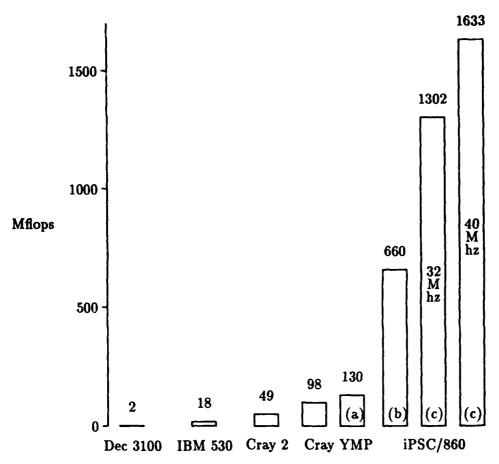


Figure 2: Performance in Mflops of KKR-CPA Code on various computers.(a) extra vectorization employed.(b) fortran only. (c) using assembly zaxpy.

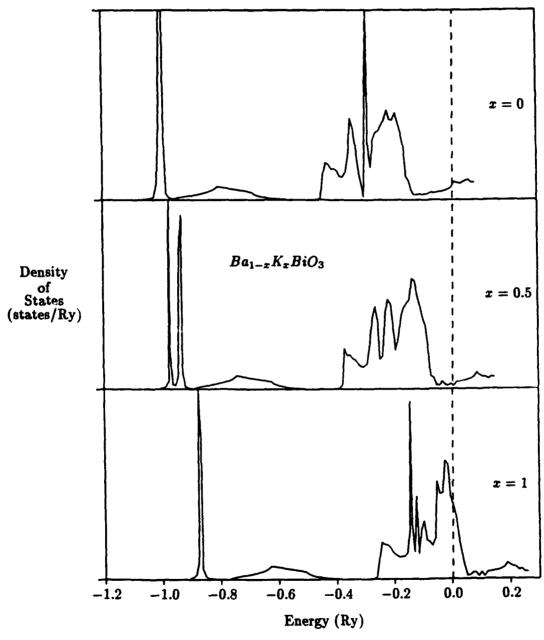


Figure 3: Affects of alloying on the density of states for $Ba_{1-x}K_xBiO_3$.

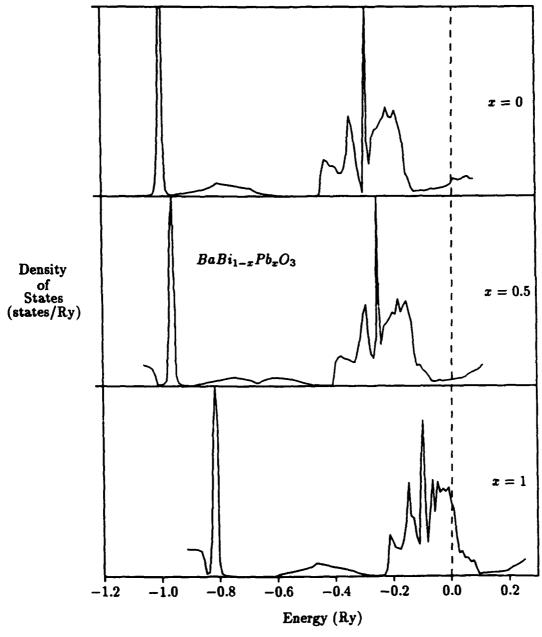


Figure 4: Effects of alloying on the density of states for $BaBi_{1-x}Pb_xO_3$.

Parallel Solutions to the Phase Problem in X-Ray Crystallography

N. Bashir ¹	M. Crovella ²	G. DeTitta ^{3,5}	F. Han ⁶
H. Hauptman ^{3,7}	J. Horvath ⁴	H. King ⁵	D. Langs ³
R. Miller ¹	T. Sabin ¹	P. Thuman ¹	D. Velmurugan ³

Graduate Group in Advanced Scientific Computing
226 Bell Hall
University at Buffalo
Buffalo, NY 14260
miller@cs.buffalo.edu

¹ Department of Computer Science, University at Buffalo, Buffalo, NY 14260

- ³ Medical Foundation of Buffalo, 73 High Street, Buffalo, NY 14203
- ⁴ Work performed while at Department of Computer Science, University at Buffalo. (Current address is Department of Computer Science, University of Wisconsin, Madison, WI 53706.)
 - Department of Chemistry, University at Buffalo, Buffalo, NY 14214
 The Upjohn Company, Kalamazoo, MI 49001

Abstract

The central problem of single crystal molecular structure determination via X-ray diffraction is the "phase problem." Associated with each diffraction maximum (a reflection) are a magnitude, which can be experimentally determined, and a phase angle, which is lost in the experiment. The goal of "direct methods" is to mathematically reconstruct the phase information from the magnitude information alone.

Traditional direct methods are capable of determining structures of moderate complexity, but to extend them to problems of the size of macromolecules (proteins, etc.) requires developing new techniques that appear to be computationally intensive. Recently, a new formulation of the phasing process, dependent on a minimal function, has been proposed. Here we explore a number of different implementations of the principle to the solution of small molecular structures. The machines that we use include

an Intel iPSC/2 hypercube, the Connection Machine CM-2, and a network of Sun workstations.

1 Introduction

A mainstay of modern structural chemistry is the single crystal X-ray diffraction technique of structure determination. This technique provides a three dimensional mapping of the positions of atoms in crystals, thereby securing unambiguous information about the architecture of molecules. The technique is robust in the sense that solids as diverse as silicon and virus crystals can be, and are, subjects fit for study. The three stages of an X-ray diffraction experiment are

1. the growth of suitable single crystals of the substance to be studied,

² Work performed while at Department of Computer Science, University at Buffalo. (Current address is Department of Computer Science, University of Rochester, Rochester, NY 14627.)

⁷ School of Medicine and Biophysical Sciences, University at Buffalo, Buffalo, NY 14214

- 2. the measurement of X-ray diffraction data, and
- 3. the unraveling of the molecular structure that agrees with the diffraction data.

The last step is frequently computationally intensive. In the experiment, a beam of X-rays of well defined wavelength, say 1 Angstrom, is trained on the crystal. The crystal is oriented so that an individual diffracting plane is brought into the Bragg condition and diffracted photons are counted either electronically or recorded photographically. The process is repeated anywhere from a few hundred to a few million times, depending on the size of the structure to be determined, as individual diffracting planes are brought into the Bragg condition. Each condition for diffraction, called a reflection, is characterized by a location on a three-dimensional grid, or reciprocal lattice, corresponding to the orientation of the crystal and the angle which the diffracting plane makes with the incoming X-ray beam. As the grid constitutes a true lattice, each reflection can be labeled by three integers, the Miller indices, that denote the location of the reflection on the reciprocal lattice relative to a common origin. Each reflection is additionally characterized by a diffraction intensity related simply to the number of counts recorded electronically or to the blackness of film recorded photographically. The intensity is related to the efficiency with which a Bragg plane diffracts X-rays. As electrons are the media that diffract X-rays, and atoms are made up of electrons centered about their nuclei, the intensity of an individual reflection is related to the density of electrons in the near vicinity of the Bragg plane. The mathematics that relates the underlying atomic arrangement in a crystal to the intensities and locations of the Bragg reflections is a three-dimensional Fourier transformation.

It would seem that all the tools necessary to unravel the structure of molecules in crystals are assembled once the diffraction experiment is concluded. Nature, however, has its own agenda. Missing, and presumably lost, in the experiment are the phases of the Fourier coefficients relative to a common reciprocal lattice origin. That is, the experiment yields the amplitudes and orientations of the Fourier components but not their phases. What nature conceals, the direct methods of structure determination seek to supply.

2 The Phase Problem

It is the determination of the set of phases, one for each reflection, that constitutes the phase problem. Early analyses of the problem led some to believe that

the problem was in principle unsolvable. An infinity of Fourier transformation maps could be had that fit the experimental results; they would differ only in the set of phases used to reconstruct the atomic arrangement. On the other hand, common sense held that since a small number of structural arrangements had been ascertained by a trial and error method there must be a solution to the phase problem. Two physical constraints make the problem not only solvable but in principle greatly overdetermined. One is the hard constraint that for a Fourier transformation to be physically meaningful it must lead to a map in which the calculated electron density (electrons per cubic Angstrom) is everywhere non-negative. The other is a softer constraint that electron density about atoms in molecules (whether in crystals or in the gas phase) is strongly concentrated about the atomic centers (the nuclei). "Non-negativity" and "atomicity" were two important principles in the earliest formulations of direct methods.

In a direct methods attack on the phase problem, probabilistic theories are used to relate the phases, or more precisely certain linear relationships among the phases, to the measured intensity data. For example, it can be shown that the sum of three phases

$$\phi_{\mathbf{H}} + \phi_{\mathbf{K}} + \phi_{-\mathbf{H}-\mathbf{K}} = \phi_{\mathbf{T}},$$

where **H** and **K** are reciprocal vectors with distinct Miller indices, e.g., $\mathbf{H} = \{3,1,2\}$, $\mathbf{K} = \{-4,3,-8\}$, and $-\mathbf{H} - \mathbf{K} = \{1,-4,6\}$, has a most probable value of 0 mod 2π radians and that probability increases as the product of the magnitudes of the intensities of the reflections $\{3,1,2\}$, $\{-4,3,-8\}$ and $\{1,-4,6\}$ increases. Such a relationship among three phases is called, in the trade, a "triple" relationship. Analogously, a "quartet" of the form

$$\phi_{\mathbf{L}} + \phi_{\mathbf{M}} + \phi_{\mathbf{N}} + \phi_{-\mathbf{L}-\mathbf{M}-\mathbf{N}} = \phi_{\mathbf{O}},$$

where the main terms L, M, N, and -L-M-N are associated with large intensities and the cross-terms L+M, M+N, and N+L are associated with small intensities has a most probable value of π mod 2π and that probability increases as the main terms become larger and/or the cross-terms become smaller. With these tools in hand a number of strategies evolved to "solve the phase problem" for small to moderate sized (up to 315 atoms at last count) structures. It is the extension of these methods to larger structures that we direct our attention.

3 The Minimal Function

As structures become larger, estimates for the phase sums (the "triples" and "quartets") become increas-

ingly less reliable. Consequently, a direct attack that is promising for small structures becomes untenable for large structures. On the other hand, whereas the number of reflections grows more or less linearly with the size of the structures, the number of phase sums explodes catastrophically. A global procedure to make the sheer numbers of phase sums work for the crystallographer exploits the probabilistic nature of the estimates of the phase sums in the following way. As the structure gets ever larger the estimate for any individual triple or quartet becomes less and less reliable, but averaged over the ever increasing number of such phase relationships, the estimates as a whole get better. A breakthrough in the use of the estimates came when it was realized that a particularly simple function of the phases, defined below, is a minimum when the correct set of phases is used to compute the function. It was quickly realized that the conceptual problem of phasing procedure was replaced by one of computational strategy.

3.1 Theory

We assume a crystal structure S in the space group G to be fixed, but unknown. The normalized structure factor magnitudes |E| are also assumed to be known. The function to be minimized is defined initially as a function, R(I), of the structure invariants:

$$R(I) = \frac{1}{D} \left\{ \sum_{\mathbf{H}, \mathbf{K}} A_{\mathbf{H}\mathbf{K}} \left\{ \cos T_{\mathbf{H}\mathbf{K}} - \frac{I_1(A_{\mathbf{H}\mathbf{K}})}{I_0(A_{\mathbf{H}\mathbf{K}})} \right\}^2 + \right.$$

$$\sum_{\mathbf{L},\mathbf{M},\mathbf{N}} B_{\mathbf{LMN}} \left\{ \cos Q_{\mathbf{LMN}} - \frac{I_1(B_{\mathbf{LMN}})}{I_0(B_{\mathbf{LMN}})} \right\}^2 \right\}, \quad (1)$$

where

$$D = \sum_{\mathbf{H}, \mathbf{K}} A_{\mathbf{H}\mathbf{K}} + \sum_{\mathbf{L}, \mathbf{M}, \mathbf{N}} |B_{\mathbf{LMN}}|.$$

It should be noted that B_{LMN} can take on negative values (see below) when the cross-terms are very small, so it sums into the denominator D as its absolute value.

$$T_{\mathbf{HK}} = \phi_{\mathbf{H}} + \phi_{\mathbf{K}} + \phi_{-\mathbf{H}-\mathbf{K}} \tag{2}$$

is the triplet,

$$Q_{LMN} = \phi_L + \phi_M + \phi_N + \phi_{-L-M-N}$$
 (3)

is the quartet,

$$A_{HK} = \frac{2}{N^{\frac{1}{2}}} |E_{H} E_{K} E_{H+K}|,$$
 (4)

$$B_{\mathbf{LMN}} = \frac{2}{N} |E_{\mathbf{L}} E_{\mathbf{M}} E_{\mathbf{N}} E_{\mathbf{L+M+N}}| \left(|E_{\mathbf{L+M}}|^2 \right.$$

$$+|E_{\mathbf{M}+\mathbf{N}}|^2+|E_{\mathbf{N}+\mathbf{L}}|^2-2$$
, (5)

N is the number of atoms, assumed identical, in the unit cell, and I_1 and I_0 are the Modified Bessel Functions. In view of Eq. (2) and (3), Eq. (1) also defines R as a function, $R(\Phi)$ of the phases ϕ . Since the magnitudes |E| are planed to be known, the functions R(I) and $R(\Phi)$ are known.

Next, the phases ϕ are themselves functions, for fixed choice of origin, of structures T,

$$E_{\mathbf{H}} = |E_{\mathbf{H}}| exp\left(i\phi_{\mathbf{H}}\right) = \frac{1}{N^{\frac{1}{2}}} \sum_{j=1}^{N} exp\left(2\pi i \mathbf{H} \cdot \mathbf{r}_{j}\right) \quad (6)$$

where \mathbf{r}_j is the position vector of the atom labeled j. Since the structure invariants $T_{\mathbf{HK}}$ and $Q_{\mathbf{LMN}}$ are uniquely determined by the structure T, independently of the choice of origin, it follows that Eq. 1 also defines a function, R(T), of structures T. Furthermore, since the magnitude of any structure invariant is the same for T and its enantiomorph, but has opposite signs for the enantiomorphs, and since only the cosines of the structure invariants appear in Eq. (1), it follows that R has the same value for T and its enantiomorph.

The minimal principle states that

$$R(S) < R(T) \text{ if } T \neq S.$$
 (7)

Since, in general, the number of phases exceeds by far the number of independent atomic coordinates needed to fix the crystal structure, a large number of identities must be satisfied by the phases. Thus the phases are not independent variables and the minimum of $R(\Phi)$, regarded now as a function of independent phases, will in general be less than R(S) but will yield values of the phases somewhat different from the true phases corresponding to the structure S. What is needed then is the global minimum of $R(\Phi)$ subject to the constraint that all identities among the phases, which must of necessity be fulfilled, are in fact satisfied.

One may go back to Eq. (1) and observe that, since the number of structure invariants exceeds by far the number of phases, and since the phases themselves are not independent, a large number of identities among the structure invariants T_{HK} and Q_{LMN} must also be satisfied. Thus the minimal principle may be reformulated as follows: Among all structure invariants T_{HK} and Q_{LMN} which satisfy the necessary identities, those invariants which correspond to the structure S minimize the function R(T). Alternatively, among all phases ϕ which satisfy the necessary identities, those corresponding to the true structure S minimize $R(\Phi)$; or finally, the (N atom) structure T which minimizes R(T) coincides with S.

4 Computational Initialization

4.1 Invariant Generation

The triplets and quartets which serve as input to our program are defined as follows. Suppose we are given n sets of Miller indices, $M_1...M_n$, where each set consists of three integers (x, y, z) which refer to the location of the reflection on the reciprocal lattice relative to a common origin. Associated with each Miller index M_i , $1 \le i \le n$, is a diffraction intensity $|E_i|$. Let a triplet t = (h, k, l) refer to the h^{th} , k^{th} and l^{th} sets of Miller indices such that $M_h + M_k + M_l = 0$. Similarly, let a quartet be defined as Q = (h, k, l, m), where $M_h + M_k + M_l + M_m = 0$

For the molecular structure that we are currently working with, we generate the triplets and quartets as follows. Sort the sets of Miller indices that were experimently determined into decreasing order by their intensity values and select the top n sets, where n refers to the number of phases to be determined. For each Miller index M_h , consider $M_k = (x, y, z)$, where $1 \le h < k \le n$, for all permutations of $(\pm x, \pm y, \pm z)$. Determine if there exists a Miller index M_l , $k < l \le n$, such that M_l , or any permutation of M_l , is equal to $-M_h - M_k$. Such an M_l may not exist if its associated intensity value, $|E_l|$, is too small. If such an M_l does exist then the triplet t = (h, k, l) is formed. See Figure 1. A similar process is followed for the quartets.

4.2 Computing R

Using the above definitions of triplets and quartets we can define ϕ_t as

$$\phi_t = \phi_{\mathbf{HK}} = \phi_h + \phi_k + \phi_l,$$

where t is a triplet equal to (h, k, l). Using this notation the calculation of R can be described as follows:

$$R = \frac{\sum_T W_T (\cos\phi_T - I_T)^2 + \sum_Q W_Q (\cos\phi_Q - I_Q)^2}{\sum_T W_T + \sum_Q W_Q},$$

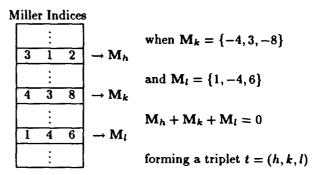


Figure 1: Example Formation of a Triplet

where $I_T = \frac{I_1(A_{\mbox{\bf HK}})}{I_0(A_{\mbox{\bf HK}})},$ $I_Q = \frac{I_1(B_{\mbox{\bf LMN}})}{I_0(B_{\mbox{\bf LMN}})},$ $W_T = A_{\mbox{\bf HK}},$ $W_Q = |B_{\mbox{\bf LMN}}|,$

I is the (known) expectation value of the cosine of the corresponding structure invariant averaged over the conditional probability distribution of triplets, W is a weight factor inversely proportional to the variance, A and B are as defined previously, and T and Q represent the set of triplets and quartets, respectively.

Notice that each term of \sum_{T} and \sum_{Q} can be computed independently. Thus, R can be efficiently computed in parallel by computing partial sums and then combining them with a global sum operation. The denominator, $\sum_{T} W_{T} + \sum_{Q} W_{Q}$, is a constant which is computed during the initialization of the data set. This initialization, which is performed once per data set, also calculates the constants W_{T} , W_{Q} , I_{T} and I_{Q} .

5 Simulated Annealing

Our initial search for the global minimum focused on the use of simulated annealing[3]. Simulated annealing is a probabilistic optimization technique designed to escape local minima in search of the global minimum. Since preliminary studies indicated that the minimal function contained a vast number of local minima, the use of simulated annealing seemed well suited to our needs.

Given an optimization function f(x), where x is a configuration of the optimization problem, simulated annealing begins by first choosing a random configuration C_0 of the optimization problem, and an initial cooling parameter c. At iteration i of the algo-

Figure 2: Simulated Annealing Algorithm

rithm, configuration C_i is perturbed to produce configuration C_{i+1} such that C_{i+1} lies within the neighborhood of C_i . Let $\Delta = f(C_{i+1}) - f(C_i)$. If $(\Delta \leq 0)$ or $(e^{-\Delta \over c} > random(0,1))$, then configuration C_{i+1} is accepted as the next configuration, otherwise configuration C_i is used as the next configuration. Notice that in order to allow the optimization function to escape a local minimum, configuration C_{i+1} may be accepted even though it has a higher cost than C_i . This process is continued for a number of iterations (the length of the Markov Chain) before c is decremented by multiplying it by a parameter c, where c 1. Thus, as the algorithm progresses it becomes more difficult to climb out of a minimum. Hopefully, this allows the function to eventually settle into the global minimum.

As can be seen from Figure 2, the process of simulated annealing involves the following parameters: the colling rate (α) , the Markov chain length, the number of cooling steps, and the amount of perturbation. These parameters form what is commonly called a perturbation scheme [1].

Our first attempt to minimize R focused on exploring the reciprocal space, which is commonly called phase space. A perturbation scheme requires us to determine the following:

- The number of phases to be perturbed at each iteration.
- The amount to perturb each phase.

- The length of the Markov Chain.
- The number of cooling steps.
- The rate of cooling, α .

We implemented a wide variety of such perturbation schemes, as described below.

In selecting the number of phases to perturb we considered choosing both a random number and a constant number of phases. Since the amount of perturbation is to be chosen such that the next configuration is within a neighborhood of the previous configuration, and since we had limited knowledge of the function, we allowed the amount of the perturbations to range from 0 to 2π radians. That is, the amount of perturbation, r, for a given perturbation scheme was chosen such that $0 \le r < 2\pi$.

We considered both a constant and a conditional number of cooling steps. When implementing the conditional cooling scheme, termination of the cooling loop occurred when the present configuration (ie., the set of phases) remained unchanged for 10 cooling steps. A variety of constant Markov chain lengths were considered based on experimentation. We allowed α , the rate of cooling, to vary between .8 and .99, as the length of the Markov chain varied.

The most promising results we obtained were with a very slow cooling rate, $\alpha = .99$, combined with a small Markov chain length, and a small perturbation amount for a random subset of the phases.

We consider this process successful if the set of phases produced are within 30° to 40° of the true set. Unfortunately, simulated annealing in phase space was not producing such results. Our next approach was to use simulated annealing in atom space. Our rational for this is that working in atom space allows us to impart chemical knowledge of the structure, such as restricting the distance between two atoms to be no closer than 1.2\AA , on the problem. In addition, the number of variables for the minimal function is reduced from approximately 10N to N, where N is the total number of atoms in the structure and 10N is the approximate number of phases being considered.

In order to use atom space we must transform the atomic coordinates to a set of phases, since the phases are needed to calculate R. This calculation is called the structure factor calculation and is computationally expensive. The structure factor for each phase p is determined as follows

$$A_p = \sum_{NA} f_j \cos 2\pi (x_p \bar{x}_j + y_p \bar{y}_j + z_p \bar{z}_j)$$

$$B_p = \sum_{NA} f_j \sin 2\pi (z_p \bar{z}_j + y_p \bar{y}_j + z_p \bar{z}_j)$$

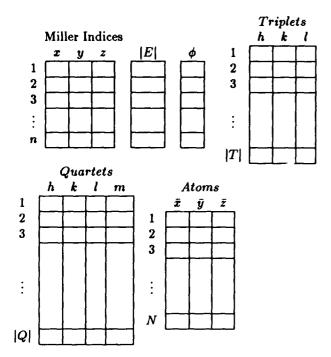


Figure 3: Data Structures

$$\phi = \arctan\left(\frac{B_p}{A_p}\right)$$

where x_p , y_p and z_p are the p^{th} Miller indices, \bar{x}_j , \bar{y}_j and \bar{z}_j are coordinates of atom j, and NA is the set of atoms including the set of symmetry elements for each atom. Thus, as can be seen in Figure 3 each Miller index, M_i , has associated with it an intensity value $|E_i|$ and a (unknown) phase, ϕ_i . Given p active processors and P phases, the calculation of the structure factor can be performed efficiently in parallel by assigning each active processor the calculation of P/p phases.

Once again a variety of perturbation schemes were attempted. We continued the use of conditional cooling lengths and allowed the cooling rate, α , to vary between $.8 \le \alpha \le .99$. Our attention then focused on determining a suitable Markov Chain length and on a method of perturbing the atoms.

Due to the nature of the structure factor calculation, the movement of one atom changes the entire set of phases. Since minimizing R is a function of the phases, we restricted the number of atoms that were perturbed at each iteration. Two main methods were implemented for perturbing an atom. In the first, we restricted the perturbation to be within a cube of edge size e, where $.5\text{\AA} \leq e \leq 6\text{\AA}$. Chemical information was used by requiring that the perturbed atom not be closer than 1.2\AA from any other atom. The second

method moved the atom in the same fashion but did not utilize chemical information about the structure. The second method generally gave lower R values but the structures produced were not necessarily feasible.

We also used two main methods for determining the length of the Markov chains. The first determined the length of each Markov chain by looking at the most recent R values within the chain. If these values did not vary by more than some ϵ , then the Markov chain was terminated [1]. The second method set the length of each Markov chain as a function of the cooling value, α . This was accomplished by allowing a Markov chain to terminate after x transitions had been accepted. Since as cooling progresses the number of acceptances decreases, the length of the Markov chain became an increasing function of the cooling value. For this reason an upper bound was placed on the length of the Markov chain [1]. The most promising results were obtained when the chain length was a function of the cooling value. Although these results were more promising than simulated annealing in phase space, we still were unable to minimize R.

6 Grid Method

It is conjectured [2] that the minimal R value of a structure with N atoms can be found by first minimizing the R value for a single atom, and then sequentially minimizing the R value of the set consisting of the previous atom(s) and one additional atom, until all N atoms have been placed. Our current strategy exploits this conjecture.

In order to gain insight into the behavior of the minimal function, we performed an exhaustive search of the three dimensional unit cell at lattice point intervals of .25Å. This showed a function which varied extremely rapidly, leading us to believe the function was much wilder than originally suspected. We then performed an exhaustive search of the three dimensional unit cell at lattice point intervals of .1Å. This search verified the unpredictability of the function. For example, within a distance of .3Å the values of R change from the highest to near the lowest. This rapid fluctuation explains why attempts to utilize simulated annealing on this function were unsuccessful.

As discussed previously, the data used for the above methods consisted of choosing the top n Miller indices by intensity and using these Miller indices to determine the corresponding sets of triplets and quartets. This data, which can be termed "high" resolution data, was originally chosen since the estimates for I_T and I_Q are more reliable. In hopes that the minimal function would become smoother we are currently us-

ing a "low" resolution data set. Our "low" resolution data set was obtained by taking the j lowest d^* values, where j is arbitrarily chosen to be greater than n. Given reciprocal lattice axes a^*, b^*, c^* and Miller index $\mathbf{M}_i = (x, y, z) \ d^*$, the length of the reciprocal vector, is equal to $a^*x + b^*y + c^*z$. These j Miller indices are then sorted in increasing order by their |E| values. The first n Miller indices of this sorted list are then used to generate the appropriate triplets and quartets. Using this "low" resolution data set we performed exhaustive searches on the three dimensional unit cell. These searches at .25Å and .1Å lattice point intervals showed the function to be much smoother than it was with the high resolution data.

For the molecular structure that we are currently working with the first atom is believed to be critical, as it selects the origin for the structure. Some structures, in different space groups, require more than one atom to be placed before the origin is determined. Therefore, we are confident that if we can determine the first atom of the structure, then we can determine the entire structure using the grid method on each successive atom. Thus, we plan on spending the bulk of our time minimizing R with respect to a single atom. We are currently focusing our attention on the question of how fine a grid is needed in order to obtain the minimum. We are also considering "grid refinement" methods that consist of multiple stages of grid applications with successively smaller grid intervals. After the grid method has been refined sufficiently for a single atom minimum, we plan on generalizing the method to larger subsets of the structure.

7 Intel iPSC/2 Implementation

On our 32 node Intel iPSC/2 hypercube, we have implemented the minimization techniques previously described in serial while the structure factor and R value calculations are performed in parallel. The reason for this is that the calculation of both the structure factor and R are computationally expensive relative to the overhead of the minimization techniques. Thus the implementation focuses on exploiting multiple processors to efficiently compute the structure factor and R value. Once we have obtained satisfactory solutions, we will consider parallelizing the minimization techniques.

The data is initially distributed so that each of the P processors have T/P triplets and Q/P quartets, plus consistent copies of all of the remaining data structures. Each processor then computes its set of T/P triplets and Q/P quartets. These partial sums are then summed to node 0 by recursive halv-

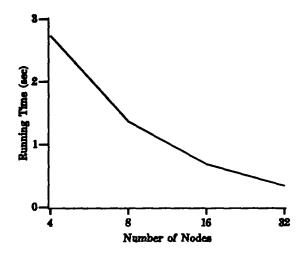


Figure 4: Time of R Calculation for a 29 Atom Structure using 300 Phases.

ing, after which node 0 performs the final division by $\sum_T W_T + \sum_Q W_Q$, which is a constant that is computed during the initialization of the data set. Notice that the calculation of any partial sum of the minimal function may require the use of the entire set of phases which is why each active node must contain a consistent copy of the entire set of phases.

This implementation evenly distributes the work and the data set among the processors. Therefore, increasing the number of processors not only allows for a faster solution, but for much larger problems to be solved. As can be seen from Figure 4, a near perfect linear speed up is observed in tests ranging from 4 (the minimum number of nodes required to hold all of the data) to 32 (the maximum number of nodes on our machine) nodes.

The calculation of the structure factor for n phases, given P active processors and a atoms, is divided into P subsets of phases. Each processor computes n/Pstructure factors. Since the computation of any one structure factor requires the use of all a atoms, each processor must maintain a consistent copy of all a atoms. In addition, all processors must maintain a complete set of phases for the calculation of R. To achieve this, the subsets of phases produced by the structure factor calculation are combined by recursive halving to node 0. Although the order of the phases within each subset will remain the same during recursive halving, the order of the subsets will not. Therefore, node 0 (the master) must order the subsets of phases prior to distributing the entire set of phases to the slaves (all active processors) by recursive doubling.

This implementation of the structure factor calculation has produced near linear speed-up, as can be seen in Figure 5. However, the speed-up of the struc-

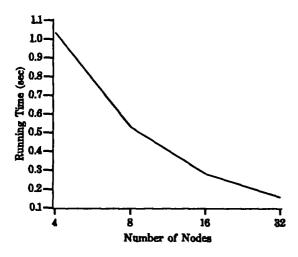


Figure 5: Time of Structure Factor Calculation for 29 Atom Structure with 300 Phases

ture factor calculation is not as efficient as that of the R value calculation, since the structure factor calculation has the additional overhead of re-ordering and broadcasting the phases.

7.1 Phase Space Simulated Annealing

For the implementation of simulated annealing in phase space on P active processors, node 0 performs simulated annealing, with each active processor cooperating in the calculation of R. This can be viewed as a master/slave implementation. At each iteration of the simulated annealing process a subset of the phases is perturbed in serial, concurrently by all processors. To insure that all processors maintain identical sets of phases, a message is broadcast to all processors, by recursive doubling, with the current random number generator seed. Since the same random number generator is used on all processors we know all processors will perturb the phases in the same manner. The master determines, by the properties of simulated annealing, whether to accept or reject the perturbed set of phases. This decision is then broadcast to the slaves by recursive doubling. To reduce the amount of message passing, the accept/reject decision and the current seed are sent in the same message.

7.2 Atom Space Simulated Annealing

A master/slave model is also used for the atom space simulated annealing implementation. Although theoretically very different, the implementations of phase space and atom space are very similar. The master performs simulated annealing, while the slaves cooperate in the calculation of R and the structure factor.

At each iteration of the simulated annealing process a subset of atoms are perturbed. A copy of the entire set of atoms on all processors must be maintained so that the structure factor calculation may be perturbed in parallel, as previously described. Therefore, to reduce the amount of message passing, we chose to perform the perturbation in serial, concurrently by all processors. We insure the perturbed set of atoms are consistent through out all processors by broadcasting the current seed to all nodes.

7.3 Grid Method

The grid method is also implemented using a master/slave model. The calculation of the structure factor and R are the same as in the simulated annealing implementations. Again each node contains the current structure and set of phases. Since we are performing a systematic exhaustive search of the three dimensional unit cell the current structure is easily maintained on all nodes.

8 Additional Architectures

In addition to the hypercube, we have implemented all three methods on the Connection Machine CM-2 at NPAC. Our current implementation requires approximately 0.2 sec. to calculate R and 0.35 sec. to calculate the structure factor for a 29 atom structure with 300 phases on 16K processors of the Connection Machine. We have also recently begun using a network of 12 Sun 4 workstations. On one workstation we can compute, in serial, R in approximately 4.3 seconds for 300 phases. We plan on using the network to divide the unit cell into 12 cells and have each workstation perform an exhaustive search of its subcell.

9 Acknowledgments

This work was partially supported by NSF grants IRI-8800514, ASC-8705104 and CHE-8508724, and by NIH grant DK-19856.

We would like to thank the Northeast Parallel Architectures Center (NPAC) at Syracuse University for allowing us to use their Connection Machine CM-2.

References

[1] Aarts, E.H.L. and P.J.M. van Laarhoven, Simulated Annealing: Theory and Applications, D. Reidel Publishing Company, New York, 1988

- [2] Hauptman, H.A. (1990) Acta Crystallographica A, in preparation
- [3] Kirkpatrick, S., C.D. Gelatt Jr. and M.P. Vecchi, Optimization by Simulated Annealing, Science, 220 pp. 671-680 (1983)
- [4] Stout, George H. and Lyle E. Jensen, X-Ray Structure Determination: A Practical Guide, John Wiley & Sons, New York, 1989

An Automata Model of Granular Materials

G. M. Gutt

Jet Propulsion Laboratory, Pasadena, CA 91109

P. K. Haff

Department of Civil and Environmental Engineering, Duke University, Durham, NC 27706

Abstract

A new modeling technique (the Lattice Grain Model) is presented for the simulation of two-dimensional granular systems involving large numbers ($\sim 10^4$ to 10^6) of grains. These granular systems (e.g., rock slides, planetary rings, industrial powders, etc.) may include both high shear rate regions as well as static plugs of grains and cannot easily be handled within the framework of existing continuum theories such as soil mechanics.

The Lattice Grain Model (LGrM) is similar to the Lattice Gas Model (LGM), which was introduced as a discrete model of fluids, in that the computation is carried out by means of cellular automata which evolve according to a simple set of rules based on local interactions. This allows large simulations to be programmed onto a hypercube concurrent processor in a straightforward manner. However, it differs from LGM in that it includes the inelastic collisions and volume-filling properties of macroscopic grains.

Examples to be presented will include Couette flow, flow through an hourglass, and gravity-driven flows around obstacles.

Introduction

Physical systems comprised of discrete, macroscopic particles or grains which are not bonded to one another occur importantly in civil, chemical, and agricultural engineering, as well as in natural geological and planetary environments. Granular systems are observed in rock slides, sand dunes, clastic sediments, snow avalanches, and planetary rings, while in engineering and industry they are found in connection with the processing of cereal grains, coal, gravel, oil shale, and powders, and are well-known to pose important problems associated with the movement of sediments by streams, rivers, waves, and the wind.

The standard approach to the theoretical modeling of multiparticle systems in physics has been to treat the system as a continuum and to formulate the model in terms of differential equations. As an example, the science of soil mechanics has traditionally focussed mainly on quasi-static granular systems, a prime objective being to define and predict the conditions under which failure of the granular soil system

will occur. Soil mechanics is a macroscopic continuum model requiring an explicit constitutive law relating, say, stress and strain; and while very successful for the low-strain quasi-static applications for which it is intended, it is not clear how it can be generalized to deal with the high-strain, explicitly time-dependent phenomena which characterize a great many other granular systems of interest. Attempts at obtaining a generalized theory of granular systems using a differential equation formalism [1] have met with limited success.

An alternate approach to formulating physical theories can be found in the concept of cellular automata, which was first proposed by Von Neumann in 1948. In this approach, the space of a physical problem would be divided up into many small, identical cells each of which would be in one of a finite number of states. The state of a cell would evolve according to a rule which is both local (involves only the cell itself and nearby cells) and universal (all cells are updated simultaneously using the same rule).

The Lattice Grain Model [2] (LGrM) we discuss here is a microscopic, explicitly time-dependent, cellular automata model, and can be applied naturally to high-strain events. LGrM carries some attributes of both particle dynamics models [3, 4] (PDM), which are based explicitly on Newton's second law, and lattice gas models [5] (LGM), in that its fundamental element is a discrete particle, but differs from these substantially in detail. Here we describe the essential features of LGrM, compare the model with both PDM and LGM, and finally discuss some applications.

Comparison to Particle Dynamics Models

The purpose of the lattice grain model is to predict the behavior of large numbers of grains (10,000 to 1,000,000) on scales much larger than a grain diameter. In this respect, it goes beyond particle dynamics calculations which are limited to no more than $\sim 10,000$ grains by currently available computing resources [3, 4]. The particle dynamics models follow the motion of each individual grain exactly, and may be formulated in one of two ways depending upon the model adopted for particle-particle interactions.

In one formulation, the interparticle contact times are assumed to be of finite duration, and each particle may be in simultaneous contact with several others [3]. Each particle obeys Newton's law, F = ma, and a

detailed integration of the equations of motion of each particle is performed. In this form, while useful for applications involving a much smaller number of particles than LGrM allows, PDM cannot compete with LGrM for systems involving large numbers of grains because of the complexity of PDM "automata".

In the second, simpler formulation, the interparticle contact times are assumed to be of infinitesimal duration, and particles undergo only binary collisions (the hard-sphere collisional models) [4]. Hard-sphere models usually rely upon a collision-list ordering of collision events to avoid the necessity of checking all pairs of particles for overlaps at each time step. In regions of high particle number density, collisions are very frequent; and thus in problems where such high density zones appear, hard-sphere models spend most of their time moving particles through very small distances using very small time steps. In granular flow, zones of stagnation where particles are very nearly in contact much of the time are common, and the hard-sphere model is therefore unsuitable, at least in its simplest form, as a model of these systems. LGrM avoids these difficulties because its time-stepping is controlled not by a collision list but by a scan frequency which in turn is a function of the speed of the fastest particle and is independent of number density. Furthermore, although fundamentally a collisional model, LGrM can also mimic the behavior of consolidated or stagnated zones of granular material in a manner which will be described below.

Comparison to Lattice Gas Models

LGrM closely resembles LGM [5] in some respects. First, for 2D applications, the region of space in which the particles are to move is discretized into a triangular lattice-work, upon each node of which can reside a particle. The particles are capable of moving to neighboring cells at each tick of the clock, subject to certain simple rules. Finally, two particles arriving at the same cell (LGM) or adjacent cells (LGrM) at the same time may undergo a "collision" in which their outgoing velocities are determined according to specified rules chosen to conserve momentum.

Each of the particles in LGM has the same magnitude of velocity and is allowed to move in one of six directions along the lattice, so that each particle travels exactly one lattice spacing in each time step. The single velocity magnitude means that all collisions between particles are perfectly elastic and that energy conservation is maintained simply through particle number conservation. It also means that the temperature of the gas is uniform throughout time and space, thus limiting the application of LGM to problems of low Mach number. An exclusion principle is

maintained in which no two particles of the same velocity may occupy one lattice point. Thus each lattice point may have no more than six particles, and the state of a lattice point can be recorded using only six bits.

LGrM differs from LGM in having many possible velocity states, not just six. In particular, in LGrM not only the direction but the magnitude of the velocity can change in each collision. This is a necessary condition because the collision of two macroscopic particles is always inelastic, so that mechanical energy is not conserved. The LGrM particles satisfy a somewhat different exclusion principle: no more than one particle at a time may occupy a single site. This exclusion principle allows LGrM to capture some of the volume-filling properties of granular material, in particular to be able to approximate the behavior of static granular masses.

The determination of the time step is more critical in LGrM than in LGM. If the time step is long enough that some particles travel several lattice spacings in one clock tick, there arises the problem of finding the intersection of particle trajectories. This involves much computation and defeats the purpose of an automata approach. A very short time step would imply that most particles would not move even a single lattice spacing. Here we choose a time step such that the fastest particle will move exactly one lattice spacing. A "position offset" is stored for each of the slower particles, which are moved accordingly when the offset exceeds one-half lattice spacing. These extra requirements for LGrM automata imply a slower computation speed than expected in LGM simulations; but, as a dividend, we can compute inelastic grain flows of potential engineering and geophysical interest.

The Rules for the Lattice Grain Model

In order to keep the particle-particle interaction rules as simple as possible, all interparticle contacts, whether enduring contacts or true collisions, will be modeled as collisions. Those collisions which model enduring contacts will transmit in each time step an impulse equal to the force of the enduring contact times the time step. The fact that collisions take place between particles on adjacent lattice nodes means that some particles may undergo up to six collisions in a time step. For simplicity, these collisions will be resolved as a series of binary collisions. The order in which these collisions are calculated at each lattice node, as well as the order in which the lattice nodes are scanned, is now an important consideration.

The rules of the Lattice Grain Model may be summarized as follows:

- 1. The particles reside on the nodes of a 2D triangular lattice, obeying the exclusion principle that no node may have more than one particle.
- 2. Each particle has two components of velocity, which may take on any value. At the beginning of each time step, each particle's velocity is incremented due to the acceleration of gravity.
- 3. The size of each time step is set so that the fastest particle will travel one lattice spacing in that time step.
- 4. Two components of a "position offset" are maintained for each particle. This offset is incremented after the velocities in each time step according to gravitational acceleration and the particle's velocity:

$$\Delta q_i = v_i \Delta t + \frac{1}{2} g_i \Delta t^2$$

where:

i = 1, 2,

 $\Delta q_i = ith$ component of increment in position offset,

 $v_i = ith$ component of particle velocity,

 $g_i = ith$ component of gravitational acceleration,

 $\Delta t = \text{current time step.}$

Once the offset exceeds half the distance to the nearest lattice node, and that node is empty, the particle is moved to that node, and its offset is decremented appropriately. Also, in a collision, the component of the offset along the line connecting the centers of the colliding particles is set to zero.

5. The order in which the lattice is scanned is chosen so as not to create a coupling between the scan pattern and the particle motions. Thus the particle position updates are done on every third

- lattice point of every third row, with this pattern being repeated nine times so as to cover all lattice sites.
- 6. Particle collisions are calculated assuming that they are smooth, hard disks with a given coefficient of restitution. Particles on adjacent nodes are assumed to collide if their relative velocity is bringing them together. The following order has been adopted for evaluating possible collisions on odd time steps: 3b, 3c, 3f, 2f, 2c, 2b, 4b, 4c, 4f, 1f, 1c, 1b; and for even time steps: 1b, 1c, 1f, 4f, 4c, 4b, 2b, 2c, 2f, 3f, 3c, 3b (where the lattice numbers and collision directions are defined in Figure 1).
- 7. In order to incorporate a container, wall, or other barrier within these rules, a second type of particle is introduced: the wall particle. This particle is similar to the movable particles, and interacts with them through binary collisions (with a separately defined inelasticity), but is regarded as having infinite mass. To allow for the introduction of shearing motion from a wall (as in a Couette flow problem), the particles making up the wall are given a common constant velocity, which is used in the usual fashion for calculating the results of collisions. However, the position of the wall particles in the lattice remains fixed throughout the simulation.
- 8. Even though a single particle does not accurately predict the trajectory of a single grain, we nonetheless regard each particle as representing one grain when we are extracting information from the simulation regarding the behavior of groups of grains. Thus, the size of one particle, as well as the spacing between lattice points, is taken to be one grain diameter.

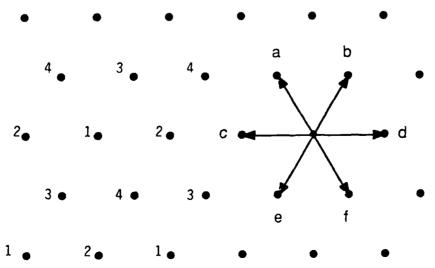


Figure 1: Definition of lattice numbers and collision directions

The transmission of "static" contact forces within a mass of grains (as in grains at rest in a gravitational field) is handled naturally within the above framework. Even though a particle in a static mass of grains may be nominally at rest, its velocity may be nonzero (due to gravitational or pressure forces); and it will transmit the appropriate force (in the form of an impulse) to the particles under it by means of collisions. When these impulses are averaged over several time steps, the proper weights and pressures will emerge.

Implementation on a Parallel Processor Computer

When implementing this algorithm on a computer, what is stored in the computer's memory is information concerning each point in the lattice, regardless of whether or not there is a particle at that lattice point. This allows for very efficient checking of the space around each particle for the presence of other particles (i.e., information concerning the six adjacent points in a triangular lattice will be found at certain known locations in memory). The need to keep information on empty lattice points in memory does not entail as great a penalty as might be thought; many lattice grain model problems involve a high density of particles, typically one for every one to four lattice points, and the memory cost per lattice point is not large. The memory requirements for the implementation of LGrM as described here are 5 variables per lattice site: two components of position, two components of velocity, and one status variable which denotes an empty site, an occupied site, or a bounding "wall" particle. If each variable is stored using 4 bytes of memory, then each lattice point requires 20 bytes.

The standard configuration for a simulation consists of a lattice with a specified number of rows and columns, bounded at the top and bottom by two rows of wall particles (thus forming the top and bottom walls of the problem space), and with left and right edges connected together to form periodic boundary conditions. Thus the boundaries of the lattice are handled naturally within the normal position updating and collision rules, with very little additional programming. (Note: since the gravitational acceleration can point in an arbitrary direction, the top and bottom walls can become side walls for chute flow. Also, the periodic boundary conditions can be broken by the placement of an additional wall, if so desired.)

Because of the nearest-neighbor type interactions involved in the model, the computational scheme was well suited to an NCUBE parallel processor. This machine consists of 512 processors, each with 512 kilobytes of memory, connected together as a 9-dimensional hypercube, along with a host computer.

For the purpose of dividing up the problem, the hypercube architecture is unfolded into a two-dimensional array, and each processor is given a roughly equal-area section of the lattice. The only interaction between sections will be along their common boundaries, thus each processor will only need to exchange information with its eight immediate neighbors. The program itself was written in C under the Cubix/CrOSIII operating system. With Cubix, only a program for the nodes of the hypercube needs be written; no separate program for the host computer is required.

Simulations

The LGrM simulations performed so far have involved from $\sim 10^4$ to 10^6 automata. Trial application runs included 2D, vertical, time-dependent flows in several geometries — Couette flow, flow out of an hourglass-shaped hopper, and flow down vertical channels with embedded obstacles.

The standard Couette flow configuration consists of a fluid confined between two, flat, parallel plates of infinite extent, without any gravitational accelerations. The plates move in opposite directions with velocities that are equal and that are parallel to their surfaces, which results in the establishment of a velocity gradient and a shear stress in the fluid. For fluids which obey the Navier-Stokes equation, an analytical solution is possible in which the velocity gradient and shear stress are constant across the channel. If, however, we replace the fluid by a system of inelastic grains, the velocity gradient will no longer necessarily be constant across the channel. Typically, stagnation zones or plugs form in the center of the channel with thin shear-bands near the walls. Shear-band formation in flowing granular materials has been analyzed earlier by Haff and others [6] based on kinetic theory models.

The simulation was carried out with 5760 grains, located in a channel 60 lattice points wide by 192 long. Due to the periodic boundary conditions at the left and right ends, the problem is effectively infinite in length. The first simulation is intended to reproduce the standard Couette flow for a fluid; consequently the particle-particle collisions were given a coefficient of restitution of 1.0 (i.e., perfectly elastic collisions) and the particle-wall collisions were given a .75 coefficient of restitution. The inelasticity of the particlewall collisions is needed to simulate the conduction of heat (which is being generated within the fluid) from the fluid to the walls. The simulation was run until an equilibrium was established in the channel (Figure 2a). The average x- and y-components of velocity and the second moment of velocity, as functions of distance across the channel are plotted in Figure 2b.



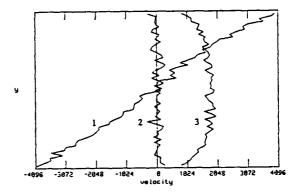
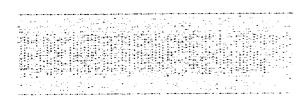


Figure 2a: Elastic particle Couette flow.

Figure 2b: X-component (1), y-component (2), and second moment (3) of velocity.

The second simulation used a coefficient of restitution of .75 for both the particle-particle and particle-wall collisions. The equilibrium results are shown in Figures 3a and 3b. As can be seen from the plots, the flow consists of a central region of particles compacted into a plug, with each particle having almost no velocity. Near each of the moving walls, a region of much lower density has formed in which most of the

shearing motion occurs. Note the increase in value of the second moment of velocity (the granular "thermal velocity") near the walls, indicating that grains in this area are being "heated" by the high rate of shear. It is interesting to note that these flows are turbulent in the sense that shear stress is a quadratic, not a linear, function of shear rate.



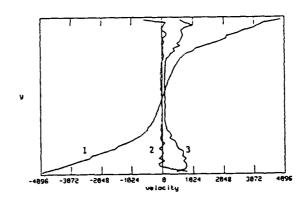


Figure 3a: Inelastic particle Couette flow.

Figure 3b: X-component (1), y-component (2), and second moment (3) of velocity.

In the second problem, the flow of grains through a hopper or an hourglass, with an opening only a few grain diameters wide, was studied; the driving force was gravity. This is an example of a granular system which contains a wide range of densities, from groups of grains in static contact with one another to groups of highly agitated grains undergoing true binary collisions. Here, the number of particles used was 8310; and the lattice was 240 points long by 122 wide. Additional walls were added to form the sloped sides of the bin and to close off the bottom of the lattice so as to prevent the periodic boundary conditions from reintroducing the falling particles back into the bin (Figure 4a). This is a typical feature of automata modeling:

that it is often easier to configure the simulation to resemble a real experiment — in this case by explicitly "catching" spent grains — than by reprogramming the basic code to erase such particles.

The hourglass flow, Figure 4b, showed internal shear zones, regions of stagnation, free-surface evolution toward an angle of repose, and an exit flow rate approximately independent of pressure head, as observed experimentally [7]. It is hard to imagine that one could solve a partial differential equation describing such a complex, multiple-domain, time-dependent problem, even if the right equation were known (which is not the case).

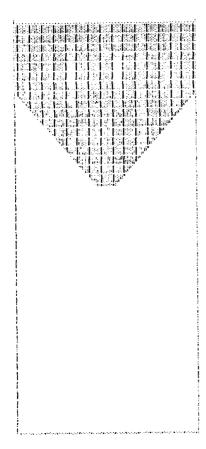


Figure 4a: Initial condition of hourglass.

Figure 4b: Hourglass flow after 2048 time steps.

Another class of problems studied involve the flow of grains around obstacles of different shapes. These flows were observed experimentally by Nedderman, Davies, and Horton [8] using a channel width of 20 cm and mustard seeds of .228 cm diameter confined between two glass plates spaced 2.3 cm apart, giving a nearly two-dimensional system. The simulation contained 16,384 particles in a lattice of 288 points by 130 points. The diameter of the circular obstacle and the side of the square obstacle were each one-half the width of the channel. Two simulations, Figures 5a and 5b, showed features qualitatively similar to those observed in the laboratory studies [8], including stagnation zones upstream of an obstacle, and void formation downstream.

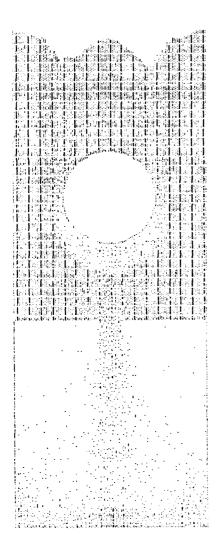


Figure 5a: Flow around a circular obstacle.

Conclusion

These exploratory numerical experiments show that an automata approach to granular dynamics problems can be implemented on parallel computing machines. Further work remains to be done to assess more quantitatively how well such calculations reflect the real world, but the prospects are intriguing.

Acknowledgments

This work was supported in part by USDOE [DE-FG22-86-PC90959] and by USARO [DAAL03-86-K-0123] and [DAAL03-89-K-0089]. We thank Geoffrey Fox for making available to us the resources of the Caltech Concurrent Computation Project and Tom Tombrello for his interest and support.

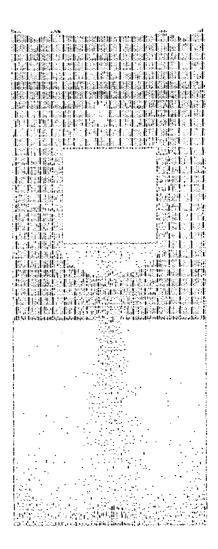


Figure 5b: Flow around a square obstacle.

References

- Johnson, P.C. and Jackson, R. (1987) "Frictional-Collisional Constitutive Relations for Granular Materials, with Application to Plane Shearing," Journal of Fluid Mechanics, 176 67-93.
- [2] Gutt, G.M. (1989) The Physics of Granular Systems, Ph.D. thesis, Caltech, May, 185p, unpublished.
- [3] Cundall, P.A. and Strack, O.D.L. (1979) "A Discrete Numerical Model for Granular Assemblies," Geotechnique, 29 (1) 47-65.
 - Walton, O.R. (1984) "Computer Simulation of Particulate Flow," Energy and Technology Review (Lawrence Livermore National Laboratory), May, 24-36.
 - Werner, B.T. (1987) A Physical Model of Wind-Blown Sand Transport, Ph.D. thesis, Caltech, April, 442p, unpublished.
 - Haff, P.K. (1987) "Micromechanical Aspects of Sound Waves in Granular Materials," Proceedings of Solids Transport Contractor's Review Meeting, (September 17-18, 1987) Pittsburgh, DOE, 41-67.
- [4] Haff, P.K. and Werner, B.T. (1987) in Colloidal and Interfacial Phenomena, 3, Particulate and

- Multiphase Processes, editors T. Ariman and T.N. Verziroglu, Hemisphere, Washington, DC, 483.
- [5] Frisch, U., Hasslacher, B., and Pomeau, Y. (1986) "Lattice-Gas Automata for the Navier-Stokes Equation," Physical Review Letters, 56 (14) 1505– 1508.
 - Margolis, N., Tommaso, T., and Vichniac, G. (1986) "Cellular-Automata Supercomputers for Fluid-Dynamics Modeling," Physical Review Letters, 56 (16) 1694-1696.
- [6] Haff, P.K. (1983) "Grain Flow as a Fluid-Mechanical Phenomena," Journal of Fluid Mechanics, 134 401-430.
 Hui, K., Haff, P.K., Ungar, J.E., and Jackson, R. (1984) "Boundary Conditions for High Shear Rate Grain Flows," Journal of Fluid Mechanics,
- 145 223-233.
 [7] Tuzun, U. and Nedderman, R.M. (1982) "An Investigation of the Flow Boundary During Steady-State Discharge from a Funnel-Flow Bunker," Powder Technology, 31 27-43.
- [8] Nedderman, R.M., Davies, S.T., and Horton, D.J. (1980) "The Flow of Granular Materials Around Obstacles," *Powder Technology*, 25 215-223.

Seismic Modeling and Inversion On The NCUBE

J. Sochacki¹, P. O'Leary², C. Bennett³, R. E. Ewing^{2,3}, and R. C. Sharpley³

¹Department of Mathematics and Computer Science, James Madison University

²Institute for Scientific Computation, University of Wyoming

³Department of Mathematics, University of South Carolina

Introduction

The uncovering of the earth's interior encompasses two important procedures. The first step is to make a prediction of the earth's interior based on scientific data collected from geophones and a knowledge of the geology of the area in which the data is being collected. The second step is to improve this prediction by numerical simulation of the wave equations. The former step is referred to as the inverse problem and the latter as the forward problem. Both problems are difficult and interconnected. Once there is scientific confidence that the structure of the interior of the earth is adequately known, this structure is encoded and analyzed under various stresses, strains, and pressures. Using wave equations to simulate these types of problems involves an immense number of calculations, overburdening the largest available vector and parallel computers. In this paper, we discuss these aspects and indicate how the distributed memory computations can be used to solve them in an efficient manner.

The inverse problem in which we are interested is to determine not only the depth, but also the shape, of complex structures below the surface of the earth. That is, given a known disturbance of the earth and the record of the geophones caused by this disturbance, we wish to accurately describe the structure. The forward problem then takes this structure and the disturbances as its data and produces the wave field over time, testing for comparison with surface waves and geophone readings. In an iterative manner, the inverse solver utilizes these results to improve the initial guess of the underlying structure of the earth. This process is repeated until convergence to the true structure is obtained to within a specified degree of accuracy. Once this accurate structure is obtained, the forward solver is used to test how this section of the earth reacts to various pressures, stresses, and strains.

Many of the inherent problems encountered in numerically approximating the wave equations to simulate the propagation of sound waves in the earth's interior have been well documented and analyzed in the literature. In this paper, we only discuss a few of the major difficulties. One of the most commonly mentioned problems is related to the vastness of the earth's interior. The numerical solution of the wave equation requires the placing of grid blocks over a finite region and therefore requires boundary conditions for the computational domain. However, the earth's interior (for the problem of interest) has no subsurface boundaries. Therefore, the boundary conditions imposed must model a 'void' boundary resulting in what are known in the literature as absorbing or radiating boundary conditions. We incorporate absorbing boundary conditions and load-balancing in the distributed memory setting of computation. Since memory is limited, the sizes for the grid blocks are bounded from below. The hyperbolic nature of the model equations requires that a correspondingly small time step be used to avoid dispersion and numerical instabilities. The limit on the grid size also restricts how accurately the interfaces of the complex structures can be represented. In this paper, we consider all these problems in the forward model using finite differences and distributed memory computing, and strongly argue that the improper treatment of any of the above-mentioned topics can lead to gross misinterpretations in the inverse problem.

The equations that we model for the pressure distribution are the acoustic wave equations

$$P_t + c^2 \vec{\nabla} \cdot (\rho \vec{v}) = 0$$

$$\vec{v}_t + \rho^{-1} \vec{\nabla} P = F(\vec{x}, t),$$

where $\vec{x} = (x_1, x_2, x_3)$, P is the pressure distribution, $\vec{v} = (v_1, v_2, v_3)$ is the velocity, ρ is the density, and c is the speed of sound in the medium. However, we actually solve the equivalent potential equation

$$U_{tt} + A(\vec{x}) U_t - c^2 \vec{\nabla} \cdot (\vec{\nabla} U) = g(\vec{x}, t),$$

where $g = c^2 \vec{\nabla} \cdot (\rho G)$ and $G_t = F$, noting that $P = -U_t$ (cf. Sochacki et al., 1990). The parameters ρ and c are determined by the structure of the earth's interior and are thus obtained from the inverse problem.

For measuring the displacement at the earth's surface, we use the elastic wave equations

$$\rho U_{tt} + A(\vec{x})U_t - \vec{\nabla} \cdot \vec{S}_1 = F_1(\vec{x}, t)$$

$$\rho V_{tt} + A(\vec{x})V_t - \vec{\nabla} \cdot \vec{S}_2 = F_2(\vec{x}, t)$$

$$\rho W_{tt} + A(\vec{x})W_t - \vec{\nabla} \cdot \vec{S}_3 = F_3(\vec{x}, t),$$

where $\vec{S}_i = (S_{i1}, S_{i2}, S_{i3})$ and $S_{ij} = \delta_{ij} + 2me_{ij}$, and δ_{ij} is the Kronecker function. Moreover, $e_{ij} = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$ and $u_1 = U$, $u_2 = V$, $u_3 = W$, $p = \sqrt{\frac{l+2m}{\rho}}$ is the p-wave velocity and $\sigma = \sqrt{m/\rho}$ is the s-wave velocity. (cf. Ewing, Jardetzky, and Press(1957)). The parameters p, σ , and ρ are also determined by the earth's structure.

The term $A(\vec{x})$ is used for the absorbing boundary conditions rather than dissipation. This term is equal to zero in the interior, since we are modeling nondissipative waves, and is assigned values on the boundary of the model so that waves are sufficiently decayed to reduce the amplitude of the spurious refelected waves off of the boundary (cf. Sochacki et al., 1987). The source terms are localized disturbances occurring either in the interior or at the surface.

Although the problems discussed above occur in both two- and three-dimensional wave propagation, we address two dimensions in this paper because of the simplicity of visualization. Also, since the problems discussed above are similar in acoustic and elastic wave propagation, we only consider two-dimensional acoustic wave equations. ever, we also discuss the problems specific to threedimensional wave propagation and elastic waves as they arise. Also, a single complicated interface can illustrate all the problems that occur with a region containing many complicated interfaces; thus the model we analyze deals with a single interface. The technique used to handle the numerical calculations required at an interface is taken from Sochacki et al. (1990). The distributed memory computation allows two different programming strategies to be considered when solving the finite difference equations. We discuss the pros and cons of these two strategies and present timings for each.

Of course, all the considerations discussed above must be displayed visually in order to allow proper analysis and interpretation. The graphics can be done on the parallel processing machine or the data sets for the graphics can be transferred to a graphics workstation that has greater visualization capabilities.

The graphics produced by the NCUBE/ten are a result of information on the nodes being dumped to an 8-bit or 24-bit graphics board. The viewpoint of these snapshots would be fixed. Hence, one can only analyze a given data set until this data is replaced by an updated set. The interactive capabilities would be acheived only through a sequence of runs. On the other hand, if data sets are passed through the Sun4 to a high performance graphics workstation via efficient data compression techniques, the process of interactive analysis is enhanced.

The Interface Problems

The model considered is 1600 meters by 1600 meters and contains a complicated interface at an average depth of 800 meters (see Figure 1). The interface is actually at a depth of 800 meters at each horizontal 8 meter interval. Between these points, the interface has a complicated and random shape. The purpose of this choice of interface configuration is to show that extremely different surface seismograms are generated by using different grid sizes. The simulations (Table 1) are run for constant size square grids varying from h=8 meters down to 2 meter. The p-wave velocity above the interface is 2000 m/s and the density is 3200 kg/m^3 . Below the interface, the p-wave velocity is 6000 m/s with a density of 2600 kg/m^3 . In all cases, the source is located at a depth of 400 meters and a horizontal distance of 800 meters. The source is the derivative of the Gaussian and has the form

$$f(t) = A(t - t_0) e^{-s(t - t_0)^2}$$

For stability, the time step Δt must be chosen to satisfy the CFL condition: $\Delta t \leq \frac{h}{c\sqrt{2}}$, where c is the maximum p-wave velocity. To minimize dispersion, the source should act for $t=2t_0$ seconds and h should be no larger than $\frac{at}{10}$, where a is the minimum p-wave velocity, and σ should be no smaller than $20 \ln(10)/t^2$. In Table 1, we give the parameters used in the model runs to show the discrepancies of surface seismograms. The time shown is the number of time steps it takes for the wave to reflect off the interface and create a reasonable surface seismogram; this is approximately .5 seconds.

Of course, the memory needed and the number of calculations done are directly proportional to the number of grids and time steps. For $h \leq 4$ it is clearly seen that a supercomputer is needed to carry out the calculations. Therefore, the parallel computer is an excellent machine to display the importance of representing an interface accurately for the inverse problem.

In the field, seismologists use source frequencies from 2 Hz to 100 Hz. In these models, a derivative of a Gaussian source provides a frequency equal to $\frac{2}{t}$. As illustrated in Table 1, a higher frequency is possible as smaller values of h are allowed. A parallel distributed memory architecture provides the memory and computational power needed to decrease h to realistic physical values. In addition, with the decrease in h, σ is used to maintain continuity in time and not for dispersion.

All the above models use a single interior source which requires that the node whose grid points contain the source information does substantially more computations on startup than the other nodes, resulting in load imbalance. This increased load is insignificant, however, compared with the total number of computations that must be done for large problems, i.e. small h. Also, to highlight the differences in the seismograms and snapshots, the data is compressed to be outputted every 8 meters so that the sizes remain the same in all three runs. Currently, the damping (or ABC) term A(x) is nonzero only for the 30 outer grid points of the bottom and edges. Hence, in the interior of the model, there is a memory drain in our algorithms. One could play off memory versus performance on this aspect, but we do not address this in the current investigation.

In both the seismograms and the snapshot for the entire wavefield it is easily seen that completely different information is given by using the finer grid sizes. It is also worth noting that the reflections off the random interfaces shows up clearly in the seismograms (Figure 2) and on the interface in the snapshots (Figure 1). Initially, however, it appears that the seismograms are similar. Therefore, the importance of having accurate forward solvers to test for interfaces in the inverse problem is highlighted in the seismograms. The memory capabilities and computational speed of the NCUBE/ten were necessary for carrying out this numerical experiment.

Computing Strategies

There are two basic methods for locating the interfaces using the finite difference scheme presented in Sochacki et al. (1990). The strategies depend on how one describes the p-wave velocity and density at each

grid point. The two different strategies arise when attempting to pass these parameters to the equations being calculated in the most efficient manner for the distributed memory. One method (Method A) is to create a matrix that contains an integer for each grid point indicating its region. Each p-wave velocity and density is assigned the corresponding integer. This means that we need a matrix equal in size to the number of grid points and two vectors equal in size to the number of structures. This strategy leads to a load balancing problem at the interfaces, since for this scheme the number of calculations away from an interface is much smaller than at the interface. For models that are not too complicated, this problem can be alleviated by strategic assignment of the nodes to the interfaces. Also, each grid point must be tested to see if it lies on an interface at each time step. This, however, does not cause load balancing problems; it just increases the number of operations.

The second method (Method B) is to create two matrices both of which are equal in size to the number of grid points. One matrix contains the p-wave velocity at each grid point while the other contains the density at each grid point. This strategy eliminates load balancing problems, because at each grid point the same calculations are being performed. However, this scheme increases the amount of memory needed and the total number of calculations done significantly. However, if there is a large number of interfaces the number of calculations is balanced by the testing of each grid point in the former scheme.

We have presented timings for both of these schemes applied to the model with the single interface for three grid sizes. In Table 2 we see that Method B is much more efficient for smaller matrices. This is due to the fact that the extra number of calculations in this method for smaller problems does not overtake the grid point checking of Method A and that the memory requirements are still minimal. However, we see that as the model size increases the calculation times for Method A and Method B approach the same magnitude. In addition, we present timings for both schemes applied to a model containing seven structures with relatively complicated interfaces to test for the trade-off in this situation between performing conditional and computational instructions (see Figure 3). The data for the various p-wave speeds (m/s)and corresponding densities (kg/m^3) of the structures are provided in Table 3.

The time increment Δt is .00047 and a source is used which has a duration of t=.126 and a spread $\sigma=3143$. We have presented timings for this medium comparing both Methods A and B in Tables 4-5. Table 4 gives the timings which include

the initial startup computations, including the source calculations, while Table 5 gives the timings for later time only.

These two tables provide a test for the trade-off between performing conditional and computational instructions in this situation. Here again, we see that the calculation times are of the same magnitude. However, in all the cases Method B is faster than Method A, and this suggests that a clever method for assigning the nodes to the interface calculations is appropriate.

We also note that in three dimensions, the sizes of the matrices for these strategies are increased in size by a factor of the number of grid points in the extra dimension; additionally, for elastic wave simulation, a matrix for the s-wave velocity is needed in the latter strategy.

Conclusions

The locating of interior structures in the earth's interior is one of the important challenges of geophysics. One method of attacking this problem is using the acoustic and elastic wave equations in two and three dimensions on distributed memory machines. In this paper we have presented two methods for accomplishing this and have presented data from these two methods performed on an NCUBE/ten. There are many more tests that can be run on the two strategies presented here, and these need to be carried out; however, the groundwork has been layed.

The data we have presented are for 2D acoustic wave analysis, but the ideas can be carried to 2D elas-

tic wave analysis and 3D in a similar manner. The main problem in 3D is that the structures become more complicated and the number of calculations is greatly increased. However, the work done here is currently being extended to 3D, and the major differences are in visualization and the fact that all the nodes of the NCUBE/ten must be used.

Acknowledgments

The authors would like to thank Chuck Baldwin, Bill Nestlerode, and Mark Oliver for all their help in porting the codes to the NCUBE and in producing the figures and graphs. This work was supported in part by Westinghouse, by Office of Naval Research Contract No. 0014-88-K-0370, by the Pittsburgh and Minnesota Supercomputer Centers, and by the Institute for Scientific Computation through NSF Grant No. RII-8610680.

References

- Ewing W.M., Jardetzky W.S., and Press F. (1957), Elastic Waves in Layered Media, Mc-Graw Hill.
- [2] Sochacki J., Kubichek R., George J., Fletcher R.W., and Smithson S. (1987), Absorbing Boundary Conditions and Surface Waves, Geophysics, 52, 60-71.
- [3] Sochacki J., George J., Ewing R., and Smithson S. (1990), Interface Conditions for Acoustic and Elastic Wave Propagation, Geophysics, in press.

Table 1. Media Parameters

h	Δt	number of	t	σ	source	total
		grid points	ļ		frequency	time steps
8	.00094	200×200	.252	3,131	7.95 Hz.	725
4	.00047	400×400	.126	12,415	15.87 Hz.	1700
2	.00024	800×800	.063	48,500	31.74 Hz.	2400

Table 2. Single Processor Timings

# grid pts.	Method A	Method B
10× 10	.018 s.	.0038 s.
25×25	.377 s.	.020 s.
50× 50	1.472 s.	1.033 s.

Table 3. Subregion parameters

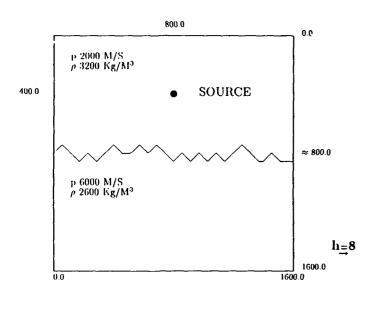
structure:	1	2&4	3&7	5&6
p-wave velocity:				
density:	1000	2800	2500	5800

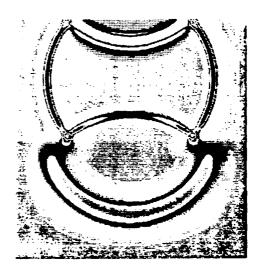
Table 4. Timings (including startup time)

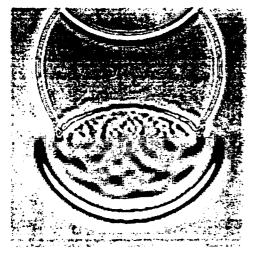
	Method A	Method B
Sequential	94.214	66.1071
Concurrent	3.0331	2.3214
Speedup	31.0678	28.4772
Percent Speedup	48.5%	44.5%

Table 5. Timings (excluding startup time)

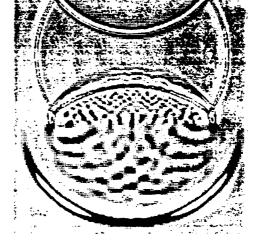
	Method A	Method B
Sequential	94.214	66.1071
Concurrent	2.7386	2.3021
Speedup	34.4084	28.7160
Percent Speedup	53.8%	44.9%







h<u>=</u>4



 $Figure\ 1.$

 $h_{\stackrel{=}{\sim} 2}$

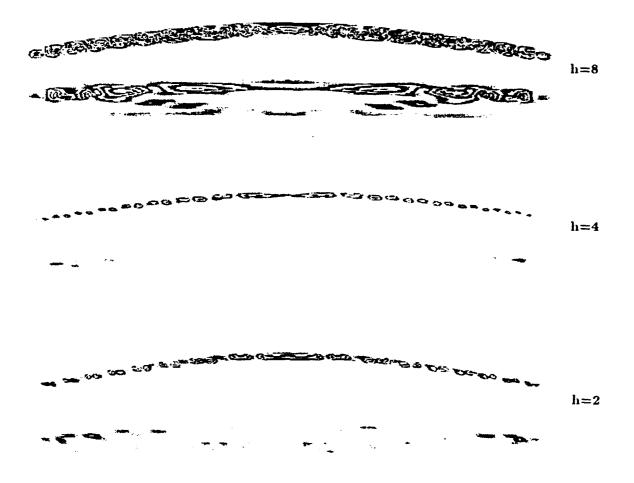


Figure 2.

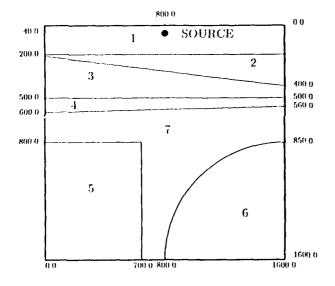
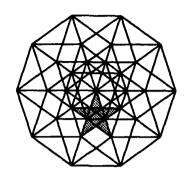




Figure 3.



The Fifth Distributed Memory Computing Conference

20: Structural Analysis

Implementation of JAC3D on the NCUBE/ten *

Courtenay T. Vaughan Sandia National Laboratories Albuquerque, NM 87185

Abstract

An implementation is presented for JAC3D on a massively parallel hypercube computer. JAC3D, a three dimensional finite element code developed at Sandia, uses several hundred hours of Cray time each year in solving structural analysis problems. Two major areas of investigation are discussed: (1) the development of general methods, data structures, and routines to communicate information between processors, and (2) the implementation and evaluation of four algorithms to map problems onto the node processors of the hypercube in a loadbalanced fashion. The performance of JAC3D on the NCUBE/ten is compared with that on a Cray X-MP: the NCUBE/ten version presently takes 20% more compute time than the Cray. On a larger simulation which used more of the NCUBE's memory, the NCUBE/ten would take less compute time than the Cray. Current activity on the newer NCUBE 2 hypercube is summarized which should lead to an order of magnitude improvement in run-time performance for the massively parallel solution of structural analysis problems.

Introduction

In this paper we discuss the implementation of JAC3D, a three dimensional finite element code which uses a nonlinear Jacobi preconditioned conjugate gradient method to solve large displacement, large strain, temperature dependent, and nonlinear material structural analysis problems, on a massively parallel computer, the NCUBE/ten hypercube. This code was developed at Sandia National Laboratories where it uses several hundred hours of Cray time each year. We note that the hypercube implementation is complete in that a user has the same user interface and simulation options on the Cray and the hypercube.

Two major implementation issues are discussed below. The first is the development of routines to communicate information between the node processors and between the host and the node processors. The reason these are nontrivial is that the finite element mesh is not necessarily regular or regularly numbered. Routines are included that determine what information each processor sends or receives at each communication step and with which processors it is communicating. The second area is the development of algorithms to map a problem onto the node processors of the hypercube in a load-balanced fashion. We will present and compare several mapping methods that, to date, have been executed on a SUN workstation.

Compute times are within 20% of the Cray X-MP for a production simulation with 89,043 equations. The NCUBE/ten can easily handle a problem four times larger; such a simulation would be faster on the NCUBE/ten relative to the Cray. Preliminary benchmarks on the NCUBE 2 indicate that the SUN front end reduces I/O time by at least a factor of ten and that NCUBE 2 processors are currently a factor of four faster than the first-generation processors. Therefore, the code should run several times faster on the NCUBE 2 than on the Cray X-MP. We are also working on parallelization of selected mapping methods and on a system to display JAC3D results from the NCUBE 2 hypercube on a Stellar graphics workstation.

Implementation Issues

Overview

JAC3D is a three dimensional finite element code which uses a nonlinear Jacobi preconditioned conjugate gradient (PCG) method to solve large displacement, large strain, temperature dependent, and nonlinear material structural analysis problems [2]. The serial version of the code reads in three data files: a control file containing material constants and numbers such as the maximum number of iterations, an input file which contains the finite element de-

^{*}This work was partially supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories which is operated for the U.S. Department of Energy under contract number DE-AC04-76DP00789.

scription of the problem, and a file which gives the temperature at each node point for each load step. JAC3D then creates an output file and an additional file used for plotting the output.

In implementing JAC3D on our hypercube, an NCUBE/ten, we have added a third input file which contains the order of the hypercube being used and a mapping of the elements and nodes of the problem onto the node processors. The NCUBE/ten is a 1024 node hypercube which has 0.5 MBytes of memory on each processor.

It was necessary on the NCUBE/ten to divide the original code into a host processor code and a node processor code. (This code division can be avoided on the newer NCUBE 2 hypercube.) The node processor code corresponds to the call to the solver in the original code, while the host code handles the input and output. The host code begins by reading the input files and doing the preprocessing that is necessary on the data. When it is ready to call the solver, it allocates a hypercube of the desired dimension and starts the solver on the node processors. In this way, running the solver on the node processors is similar to calling the solver as a subroutine with the passed variables now being communicated between the host and node processors.

PCG and Finite Element Methods

The iteration matrix is calculated at each iteration as it is needed, which avoids using the memory which would be required to store the entire matrix. The matrix is calculated element by element, so some information about each of the elements has to be kept. This is done by dividing the elements among the processors such that each element is assigned to one processor. In this way, there are no duplicate calculations.

Each element has a list of nodes which are associated with it and allocates storage for all of these nodes. In this way each node may be allocated space in more than one processor but the node will be assigned to only one processor. That processor is responsible for maintaining the correct value of the variables associated with the node by collecting partial values of the variables associated with that node from other processors and providing these correct values to the other processors when needed. On each processor, the nodes which are assigned to it are numbered first, followed by the nodes for which the processor needs values but which are assigned to other processors. In this way, each processor locally numbers the nodes and elements that it has.

In the solution algorithm, the unknowns at the nodes are updated in two ways. Some calculations,

such as the calculation of the residual vector, are done element by element [6]. In order for the processors to update the unknowns associated with an element, some values of other variables at the associated nodes need to be communicated to that processor. As each element is used, the unknowns at the nodes associated with that element are updated. Since each element appears in only one processor, several processors will generate updates to shared variables, which requires communication of partial results so these updates can be combined to form the final result.

The second way that unknowns at a node get updated is by the processor which to which that node is assigned. An example of this is the calculation of the new direction vector from a linear combination of the previous direction vector and the residual vector.

Initial host-to-node Communication

The host processor communicates with the node processors by communicating only with node 0. Any data that the host sends to the node processors is sent to processor 0 which then broadcasts the information to the rest of the processors by means of a fanout algorithm using a minimal spanning tree of the hypercube rooted at node processor 0 [4]. In the fanout algorithm, successive dimensions of the hypercube are used. In each stage, all of the active processors send information to their neighbor in that dimension. As those processors receive information, they become active and will send information in the next stage.

The host processor starts by sending the node processors a message which contains startup information such as the total number of elements and nodes in the problem and the maximum number of iterations. The node processors then use this information to set up some temporary arrays. The host processor then reads in the problem mapping of the elements and sends that information to the node processors. This allows the node processors to determine how many elements they have and allocate space for some arrays. The host processor then sends the list of nodes which are associated with each of the elements and the mapping of the nodes to the node processors. The node processors store the portion of the list of nodes which are associated with their elements and then use that with the mapping of the nodes to determine the number of nodes they need storage for and to set up communication with other nodes.

Data Structures for Interprocessor Communication

Next, the node processors set up the communication which they do during the calculations. Using the list of which processor has each node, a processor constructs a list of nodes for which it needs values of variables but which are assigned to other processors. This receive list of nodes is then sorted by processor and the processor builds an index to this list consisting of the processor to communicate with, the number of nodes which have to be communicated, and a starting index into the list. This is illustrated in Figure 1. When the list is sorted by processor, it is ordered by placing the processors in descending order of their distance from the processor in terms of message hops. In this way, messages which will take the longest time to be communicated will be sent first. In our experiments, this message order cut down the execution time of the algorithm.

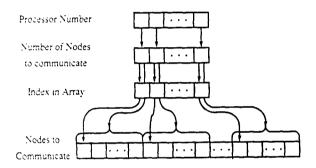


Figure 1. Communication Data Structure

Each processor sends the processors which it needs information from the list of nodes it needs from that processor. Each processor uses this information it receives to construct a list similar to its receive list, a send list which is used to send correct values of variables. The communication routines use this general data structure since the problems to be solved are generally irregular and have an irregular numbering of the nodes.

When the processors need to communicate the value of a variable, they use the send list to send messages to other processors and the receive list to receive messages from other processors. When the values of an array need to be communicated, each processor sends a message to each of the processors in its send list of processors. The processor numbers in the send list are used successively and an index into the node numbers being sent is maintained. For each processor in the list, the node numbers to be sent are determined by taking them from the list of nodes starting at the index. Since the number

of nodes to be communicated to each processor is stored, that many nodes numbers are used to take information from the array to be sent and put into a message array. This process uses all of the data structure as illustrated in Figure 1 except for the array of indexes into the list of nodes to be communicated. The message array is then sent and the index is incremented by the number of nodes which were sent.

When a processor receives a message, it looks up the processor number in its receive array and the number of nodes that are being communicated and the starting position in the array. It uses that information to put the values in the message into the variable array in the right places. Since it can be seen that the communication involved with the process of communicating correct values of variables at a node between the processors is the inverse of the the process of communicating partial values of variables at a node between processors, the receive list is used to send partial results to other processors and the send list is then used to receive those results which are added to the local results to get the correct value. In the case of communicating partial values, the final result does not necessarily need to be sent to the other processors involved since they may not need this value.

The other case in which interprocessor communication has to be done is the case of inner-products. This is done by the standard bidirectional exchanges of partial information along successive dimensions of the hypercube with the addition of partial results after each exchange [6].

Input: Large Vectors

After the node processors have allocated space for the vectors that they store and have set up their communication schemes, the host processor can send them the initial vector information(e.g. temperatures). This information is sent to processor 0 which then broadcasts it to the other node processors. Each processor then takes the part of the vector which it needs and stores it in its memory. The maximum size of a message, the size of the message buffers on the node processors, and the size of an array on the host processor are each limited, so large messages have to be read in to the host and sent to the node processors in pieces. After each piece of the message is received by the node processors, node processor 0 sends a message back to the host to allow the host to send the next piece. This procedure prevents message buffer overflow on the node processors.

Host Activity During Node Computation

At this point the node processors start calculating and the host processor waits. Since the node processors have to output results and read additional input such as the temperature of the nodes at the beginning of each load step, the host processor has to be able to call the appropriate subroutine to interact with the node processors. It does this by waiting to receive a message and, based on the type of the message received, either calls the appropriate subroutine, prints out the appropriate error message. or deallocates the hypercube and quits. Since node 0 has a copy of any scalar data which has to be communicated back to the host processor to run the subroutine, this information is included in the message which tells the host processor which subroutine to run. In summary, execution is controlled by the node processors in this part of the calculation.

Output

The output from the node processors is handled by a fanin algorithm, in which the information to be output is sent to node processor 0 which, in turn, sends the information to the host. The fanin algorithm is the inverse of the fanout algorithm. At each stage, half of the active processors send a message to the other half. The processors which receive a message are the active processors for the next stage. As with input, output of large messages is also done in pieces. In order to output arrays in the proper order, each processor has a list of the global order number of the nodes which are assigned to it. Each piece of the array is assembled in the global order and sent to the host processor.

Problem Mapping

In order to implement JAC3D on the hypercube, we had to provide for the automated mapping of large problems onto the hypercube. We have used four mapping methods. The first is a recursive bisection method developed for problems on rectangular grids by Berger and Bokhari [1]. In this method, the problem grid is divided into two rectangles along a line of the grid. This division is repeated recursively to each of the rectangles until the desired number of sets of unknowns is created. This method is easily adapted for three-dimensional rectangular grids [3]. This method has the disadvantage that it has the potential for load imbalance, since each set is divided along a line of the grid and, therefore, the two resulting sets may not be the same size.

From this algorithm, we have developed a second algorithm which uses recursive bisection for irregu-

lar regions in three dimensions. The first step is to sort the nodes of the grid in the x, y, and z directions. At each stage of the mapping, a direction is chosen and each set in the mapping is divided into two equal or nearly equal sets based on the index in the sorted list for the given direction of each node in the set. For example, given a set S with n nodes which is being divided into sets S1 and S2 along the x direction, the first n/2 nodes of set S in the sorted list of nodes for the x direction are placed in set S1 with the remainder put in set S2. In this way, the sets at the final stage of the mapping will have an approximately equal number of nodes.

The third algorithm was developed by Kernighan and Lin [8]. It is a iterative graph-based algorithm which starts with a set which has been arbitrarily divided into two equal sized pieces and exchanges nodes in order to minimize the number of edges connecting the two pieces of the set. At each iteration, it looks at all of the unmarked nodes in each of the two pieces of the set and marks the pair which, if exchanged, would minimize the number of edges connecting the two pieces. After all of the nodes are marked, then the minimum number of pairs to create the maximum change are exchanged. The process is repeated until nothing further can be gained by swapping nodes.

The fourth algorithm that we used is a graph-based algorithm developed by Vaughan [9]. At each stage, each set is divided into two equal parts by the use of level sets. The first step to divide a set into two pieces is to find a pseudo-diameter of the graph of the grid [5]. A rooted level structure is constructed from each endpoint of the pseudo-diameter. The nodes are divided into two sets according to which endpoint they are closer to. Each rooted level structure will have a set of level sets and the number of the level set a node is in is a measure of its distance from the root of the level structure. Points which are equidistant from both endpoints are assigned to a set so that the sizes of the sets are equalized.

By using the endpoints of a pseudo-diameter as starting points, we seek to construct level structures with small level sets thus providing a smaller set of nodes on the boundary when the set is divided into two pieces. This is similar to the motivation for using level structures in reordering equations for solution by direct methods.

For the two graph-based algorithms, the number of sets at each stage of the division is doubled from n to 2n and the sets are divided according to their set number in a gray code fashion. When the first set, set 0, is divided into two sets, these sets are

numbered 0 and n arbitrarily. After set i is divided, with 0 < i < n, the two resulting sets are numbered i and i+n. The choice of which set is to be numbered i is determined by which numbering gives the smallest cost for communication with the sets which have already been divided.

For each of the algorithms, the nodes are divided among the processors. However, with our solution method, the elements also have to be mapped to the processors. Each of the mappings above work by doubling the number of processors in the mapping at each stage. At each stage, half of the nodes and half of the elements assigned to a processor are assigned to a new processor. Each element stays in its processor or moves to the new processor based on which of the two processors has more of its nodes. Ties are settled in such a way as to keep the number of elements assigned to the two processors even.

Results

We solved two problems with JAC3D on the hypercube. The first is a rectilinear block with three materials, 450 elements, and 810 nodes. The second is a solder analysis problem of a 28 pin integrated circuit on a PC board. It has four materials, 22932 elements, and 29681 nodes and is very irregular (Figure 2). Since we are solving for the displacements in three directions, there are 89043 unknowns in this problem. Symmetry is used in the x and y directions to decrease problem size. Note that most of the elements and nodes are in the pins connecting the PC board to the integrated circuit.

Table 1 shows the execution times for the program on the first problem using the four mapping methods as well as a mapping constructed by hand. The problem would not fit on one processor, or even two processors in the case of the Berger and Bokhari mapping. In the tables, hand is the hand mapping, graph is the graph-based method by Vaughan, kl is the Kernighan and Lin algorithm, bb is the Berger and Bokhari algorithm, and rb is the recursive bisection method based on a modification of the Berger and Bokhari algorithm. The execution times only include the node processor time and do not include the preprocessing time for the host. In the best case, we got a speedup of 41 on going from two to 256 processors. This is encouraging considering that, on 256 processors, each processor had two or fewer elements and four or fewer nodes.

Table 2 shows the time to construct the mappings on a SUN 3. These times are smaller by a factor of two or three than the time the division would take

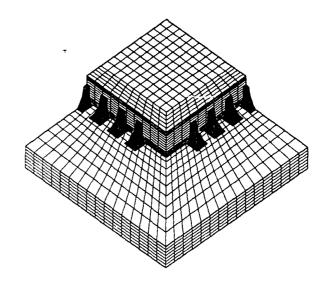


Figure 2. Solder Analysis Problem

<u> </u>	Table 1.				
E	Execution time for small problem				
Ĺ		(secor	ıds)		
cube		Divis	ion Met	hod	
dim	hand	graph	kl	bb	rb
	1747	1751	1751	-	1752
2	885	910	911	1003	909
3	463	473	468	562	486
4	252	254	258	313	271
5	139	141	150	192	145
6	86.8	84.1	108	117	85.3
7	-	60.9	65.5	82.9	57.8
8		45.9	48.0	55.8	42.9

on one node processor of the NCUBE/ten. The two graph-based methods are slowest while the Berger and Bokhari algorithm is the fastest. Note that a large portion of time for the Kernighan and Lin algorithm is spent in the first division.

Table 3 shows the execution time for the solder analysis problem on the NCUBE. The time includes all of the node time from the time the host communicates the problem to the nodes and does not include the host preprocessing time. The Kernighan and Lin algorithm produces the mapping which executes the fastest while the other two methods are about equal. As Table 4 shows, however, construction of the Kernighan and Lin mapping is the slowest by at least a factor of ten.

Table 5 compares the solder analysis problem run on both the NCUBE and the Cray X-MP. Here, compute time for the NCUBE is just the node pro-

Table 2. Mapping time for small problem (seconds on a SUN 3)				
cube		vision I		d
dim	graph	kl	bb	rb
1	3.0	30.3	2.0	2.4
2	4.3	40.8	2.0	2.5
3	6.6	46.4	2.1	2.8
4	10.4	52.5	2.1	3.1
5	14.5	58.0	2.1	4.0
6	21.5	66.3	2.2	5.4
7	29.1	74.9	2.3	7.9
8	40.0	84.0	2.4	14.3

Table 3. Execution time for large problem (seconds)				
cube	D	Division Method		
dim	kl	graph	гb	
8	8243	9098	9089	
9	5541	6217	6331	
10	4312	4602	5144	

cessor time without any of the overhead of communicating with the host between load steps, while the total time is the time from start to finish on the host. The total execution time for the NCUBE/ten including all of the host time was 6100 seconds. This shows that the processing time on the NCUBE/ten is comparable to that on the Cray X-MP but the I/O time which is a result of the host processor of the NCUBE/ten causes the total execution time on the NCUBE/ten to be much larger than that of the Cray. When we implement this code on the NCUBE 2 with the SUN front end, the ratio of the total time to compute time should improve dramatically.

Discussion and Conclusions

We have implemented a large 3D finite element code on the NCUBE/ten hypercube and have obtained supercomputer-class performance (except for

Table 4. Mapping time for large problem (seconds on a SUN 3)			
cube	Division Method		
dim	kl	graph	rb
8	49193	2995	684
9 [50072	3751	1242
10	50775	4894	2283

Table 5. NCUBE vs. Cray X-MP (seconds)		
Compute Tim		
NCUBE/ten	2197	
Cray X-MP	1661	

host processor I/O). Compute times are within 20% of the Cray X-MP for a production simulation with 89,043 equations. The NCUBE/ten can easily handle a problem four times larger; such a simulation would be faster on the NCUBE/ten relative to the Cray. The hypercube code is complete: a user sees the same user interface and simulation options on the Cray and the hypercube.

We are now implementing this code on the NCUBE 2 and its SUN front end. Preliminary benchmarks indicate that the SUN front end reduces I/O time by at least a factor of ten and that NCUBE 2 processors are currently a factor of four faster than the first-generation processors. Therefore, the code should run several times faster on the NCUBE 2 than on the Cray X-MP. We are also working on a system to display JAC3D results from the NCUBE 2 on a Stellar graphics workstation.

Several promising methods have been implemented and compared for mapping general problems onto a hypercube. Clearly, the methods should be judged by both the quality of their mappings and the time it takes to do the mapping. We plan to implement selected mapping algorithms, including the simple graph method and the recursive bisection method, in parallel on the NCUBE 2. We expect that some of the mapping algorithms will parallelize well and that the time used for mapping will ultimately be a small part of the overall execution time.

References

- [1] Berger, M. J. and Bokhari, S. H. (1985) "A Partitioning Strategy for PDEs Across Multiprocessors", in *Proceedings of 1985 Int. Conf. Par. Proc.*, pp. 166-170.
- [2] Biffle, J. H. (1984) "JAC A Two-Dimensional Finite Element Computer Program for the Non-Linear Quasistatic Response of Solids with the Conjugate Gradient Method", SAND81-0998, Sandia National Laboratories, Albuquerque, NM.
- [3] DeVries, R. C. (1990) "Static Load Balancing on a Hypercube: Concepts, Programs, and Results", SAND90-0338, Sandia National Laboratories, Albuquerque, NM.
- [4] Geist, G. A. and Heath, M. T. (1986) "Matrix Factorization on a Hypercube Multiprocessor", in Hypercube Multiprocessors 1986 (M. T. Heath, ed.), SIAM, Philadelphia, PA,

- рр. 161-180.
- [5] Gibbs, N. E., Poole, W. G., and Stockmeyer, P. K. (1976) "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix", SIAM J. Numer. Anal. 13, 1976, pp. 236-250.
- [6] Gustafson, J. L., Montry, G. R., and Benner, R. E. (1988) "Development of Parallel Methods for a 1024-Processor Hypercube", SIAM J. Sci. Stat. Comp. 9, 1988, pp. 609-638.
- [7] Jiang, B-N. and Carey, G. F. (1984) "Subcritical Flow Computation Using an Element-By-Element Conjugate Gradient Method", in *Proc. 5th Int'l. Symp. Finite Elements and Flow Problems*, Univ. of Texas, Austin, Jan. 23-26, pp. 103-106.
- [8] Kernighan, B. W. and Lin, S. (1970) "An Efficient Heuristic Procedure for Partitioning Graphs", Bell System Technical Journal, 49, pp. 291-307.
- [9] Vaughan, C. T. (1989) "The SSOR Preconditioned Conjugate Gradient Method on Parallel Computers", Ph.D. Dissertation, University of Virginia.

Porting the ABAQUS Structural Analysis Code To Run On the iPSC/2

Michael L. Barton Edward J. Kushner

Intel Scientific Computers 15201 NW Greenbrier Parkway Beaverton, Oregon 97006

Abstract

This paper describes initial efforts to convert the structural analysis program, ABAQUS, to run on the iPSC/2. Efforts were limited to the main program since it is much more demanding of computational resources than either the pre- or post-processor. The main program, in turn, can be viewed, from the perspective of parallel processing, as consisting of two steps with separate domain decompositions: the generation of submatrix data and the solution of a large system of linear equations (typically, out of core).

Parallel generation of submatrix data was achieved by distributing the individual finite elements among the processing nodes.

The equation solver used by ABAQUS is an implementation of the wave-front method. The complex nature of factorization for the wave-front method and the frequent need to do disk I/O dictated the use of a hybrid decomposition where one processor executed non-numerical operations associated with factorization while the other processors assisted the manager by performing all calculations associated with Gaussian elimination.

Introduction

ABAQUS [1] is a large commercial finite element code extensively used for structural analysis. It is written and developed for sequential computers and presents major challenges to parallel processors. In addition to the usual problems of load balancing and minimization of communication between processors, a conversion of ABAQUS must contend with an extensive file system and significant disk I/O. It is

only with the availability of the Concurrent I/O Facility as a feature of the iPSC/2 that such a conversion could be considered.

The solution of a structural analysis problem with ABAQUS is typically a three-stage process with successive execution of a pre-processor, main program and a post-processor. By far the most demanding in terms of computational resources is the main program, and it is this program that was modified to run in parallel.

An initial examination of the main program indicated that two separate decompositions would be recessary to achieve efficient performance. The first decomposition supports parallel generation of submatrix data, while the second is needed to solve the resulting system of linear equations. Such an approach is possible since matrix generation and matrix solution are decoupled processes within ABAQUS.

Parallel ABAQUS Assembly

The generation of element stiffness matrices is the most intrinsically parallel part of any finite element code. The local matrix associated with each element is calculated based on data entirely local to that element, and the potential parallelism is limited only by the access the processor has to the data associated with a given element. The element type, number of degrees of freedom, material data, element connectivity and node data must all be known to the processor in order to calculate the element stiffness matrix and right-hand-side contribution.

In ABAQUS these data are initially generated by the pre-processor and transmitted to the main program through a communication file. Since ABAQUS imposes no upper limit on problem size, none of this information is kept entirely in data arrays in the main routine. Rather, the data for a particular data base starts in an array in memory and, if it is too large to fit entirely, spills over into a data base file. In addition, a portion of memory is devoted to a pool of software controlled cache pages so that frequently-used data from the data base files may be accessed with minimum latency.

This data base for finite element analysis has been carefully designed for performance on a wide variety of single memory machines. To properly parallelize ABAQUS, the domain decomposition requires the partitioning of those data bases. The element and element operator data bases, for example, are readily split so that each processor has a data base of its own elements. The node data base is more complicated since nodes on the boundary between regions belonging to different processors must be shared. This is probably best implemented by assigning two node data bases to each processor, one for nodes internal to that processor's region, and one for shared nodes. Shared nodes would then require special processing to update the displacements after each iteration.

For the effort described in the paper, only the element and element operator data bases have been split. All others are replicated on each processor, with a special procedure required at the end of the run to bring one copy of the node data base up to date.

Parallel Assembly Performance

Since assembly is intrinsically parallel, linear speed-up is expected as more processors are employed. This speed-up should be reduced only by contention for I/O bandwidth in reading the communication file and writing to the element operator files. Table 1 shows the speed ups for the Submatrix Generation on a very large statics problem. These tests are run on an iPSC/2-SX with 8 I/O nodes and 8 disks.

Even better speed up would be expected when the data base files are properly decomposed as suggested above. This improved decomposition will reduce I/O traffic by reducing needless replication of data and

# Processors	Time (sec.)	Speed up
1	1500.1	1.0
2	746.0	2.0
4	374.0	4.0
8	188.7	7.9
16	95.2	15.8
32	53.0	28.3

Table 1 Performance Results for Submatrix Generation

by better utilizing processor memory. Additional I/O nodes would also improve performance.

Parallelization of the Equation Solver

In order to determine the feasibility of executing the equation solver in parallel, only a subset of the software from ABAQUS that addresses linear systems was modified. In particular, attention was limited to the subroutine that factors symmetric stiffness matrices. The method used by ABAQUS for matrix factorization is an implementation of the wave-front algorithm [2]. An initial examination of the implementation suggested the use of a hybrid decomposition where:

- One processor (the manager) executes the factorization routine except for numericallyintensive calculations. The other processors (the workers) then assist the manager by performing the numerically-intensive calculations. Among the tasks left to the manager are bookkeeping, stability checks and disk I/O.
- 2. The allocation of work among the workers is via a domain decomposition of the coefficient matrix of the wave-front. The domain decomposition is the standard decomposition that has been used elsewhere to solve systems of linear equations on the iPSC/2 (e.g. LINPACK); the columns of the coefficient

matrix are distributed in a round robin fashion to participating processors.

As an example to illustrate the allocation, consider a four processor system. The first processor (node #0 in the numbering scheme of the iPSC/2) executes the factorization subroutine except for numerically intensive operations. Node #1 then performs all numerical calculations associated with columns 1, 4, 7, 10... of the coefficient matrix, node #2 performs the same calculations for columns 2, 5, 8, 11... and node #3 performs in the same fashion for columns 3, 6, 9, 12.... The result is two separate programs; a manager program that runs on node 0 and a worker program that runs on nodes 1, 2, and 3.

The numerical calculations that have been transferred from the manager to the workers are all DO loops that involve two specific arrays. The first array (GPA) contains the coefficients of the equations in the wave front while the second (BBAXO) is a two-dimensional array of all equations that become fully assembled when a submatrix is added to the wave front.

The division of work and transfer of data between the manager and worker programs are depicted in Fig. 1. This division results in six communication points in the application, of which four involve communication between the manager and worker nodes.

Results for Matrix Factorization

All benchmarks were run on an iPSC/2 with SX nodes and 8 MBytes of memory per node. SX configurations can deliver as much as .5 MFLOPS per processor of computational power in double precision floating point operations. The concurrent I/O facility consisted of two I/O nodes with 2 disks on each node.

Benchmarking was done on the same large problem that was the basis of the results that were presented for the generation of submatrix data. This problem creates a linear system consisting of 39000 equations with half bandwidth of 440. Execution times are presented in Table 2.

# Processors	Time (sec.)	Speed up
1	38070	1.0
4	13639	2.8
8	7183	5.3
16	4368	8.1
32	4412	8.6

Table 2 Performance Results for Matrix Factorization

Concluding Remarks

Results presented herein indicate excellent parallel performance for the generation of submatrix data and acceptable performance for matrix factorization. With the availability of the iPSC/860, absolute performance will be improved significantly. Some initial results bear this out. Execution of submatrix generation on two i860 nodes improved the applicable results of Table 1 by a factor of 11. It is anticipated that performance for matrix factorization will be improved by a factor at least as large as this.

References

- [1] ABAQUS Manual (1987) Hibbitt, Karlsson & Sorensen, Inc. Providence, R.I.
- [2] Hinton, E. and Owen, D.R.J. (1977) Finite Element Programming, Academic Press, San Francisco.

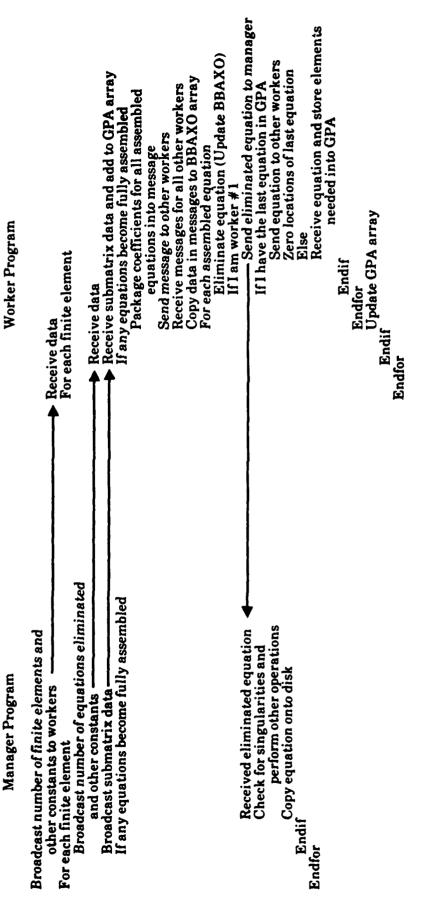
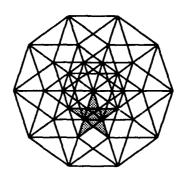


Fig. 1 Division of Effort Between Manager and Worker Programs for the Subroutine SOLZIP



The Fifth Distributed Memory Computing Conference

21: PDE Methods

Conjugate Gradient Methods for Spline Collocation Equations

Christina C. Christara
Department of Computer Sciences
University of Toronto
Toronto, CANADA, M5S 1A4

Abstract

We study the parallel computation of linear second order elliptic Partial Differential Equation (PDE) problems in rectangular domains. We discuss the application of Conjugate Gradient (CG) and Preconditioned Conjugate Gradient (PCG) methods to the linear system arising from the discretisation of such problems using quadratic splines and the collocation discretisation methodology. Our experiments show that the number of iterations required for convergence of CG-QSC (Conjugate Gradient applied to Quadratic Spline Collocation equations) grows linearly with the square root of the number of equations. We implemented the CG and PCG methods for the solution of the Quadratic Spline Collocation (QSC) equations on the iPSC/2 hypercube and present performance evaluation results for up to 32 processors configurations. Our experiments show efficiencies of the order of 90%, for both the fixed and scaled speedups.

1. Introduction.

We study the parallel solution of the Partial Differential Equation (PDE) problem

$$Lu = aD_{x}^{2}u + bD_{x}D_{y}u + cD_{y}^{2}u + dD_{x}u + eD_{y}u + fu = g$$
 (1.1)

in
$$\Omega \equiv (ax,bx) \times (ay,by)$$

$$Bu \equiv \alpha u + \beta D_n u = g_0$$
 on $\partial \Omega \equiv \text{boundary of } \Omega$ (1.2)

The input functions in the PDE problem (1.1)-(1.2) are assumed to be functions of x and y in $C^1[\Omega]$, while D_n denotes the normal derivative of u on $\partial\Omega$.

In this paper we discuss the application of Conjugate Gradient (CG) and Preconditioned Conjugate Gradient (PCG) methods to the linear system arising from the discretisation of the above problem using quadratic splines and the collocation discretisation methodology. The fact that we have used quadratic splines does not limit the importance of our results, since the use of other degree splines gives rise to linear systems with similar properties to those of the quadratic spline equations. Discretisation methods other than Spline Collocation (SC) are known to give rise to similar structure linear systems.

We implemented the CG and PCG methods for the solution of the Quadratic Spline Collocation (QSC) equa-

tions on the iPSC/2 hypercube and present performance evaluation results for up to 32 processors configurations. The implementation can be straightforward extended to several other MIMD architectures, including linear array, 2-dimensional grid of processors, as well as shared memory machines.

The methods for the parallel computation of PDEs can be classified in 3 general groups: the domain decomposition or substructuring methods, in which we assume the decomposition of the domain of problem definition into non-overlapping subdomains, the domain splitting methods, in which the domain is decomposed into overlapping subdomains, and those methods that directly use the parallelism involved in the process of solving the linear system arising from the discretisation of the PDE problem.

In [Chri90a], [Chri89], [Chri88a] we have studied domain decomposition methods for solving the above PDE problem and presented the results from the implementation of those on the iPSC/2, NCUBE/7 and SEQUENT BALANCE 21000 parallel machines. In [Hous88b] domain splitting methods are integrated with cubic spline collocation and implemented on the NCUBE/7 hypercube. This paper falls in the third category of methods for the parallel computation of PDEs.

Many researchers have studied the convergence and/or parallel implementation of CG and PCG methods applied to systems arising from the discretisation of elliptic problems by other Finite Element Methods (FEMs), or to the Schur complement systems arising from domain decomposition methods and appropriate reordering [Keye87], [Dryi84], [Bram86], [Bjor86], [Dryi86]. Others [Rodr86], [Tang87] experiment with domain splitting methods. The study of the solution of SC equations is limited due to the fact that the development of optimal schemes for two-dimensional problems is very recent [Hous87], [Irod87], [Chri88b], and due to the lack of some nice properties, such as symmetry and positive definiteness, which are often standard properties for other FEM equations. This paper is the first successful study of the application of CG methods to SC equations.

2. The Quadratic Spline Collocation method.

Spline collocation methods have been proven an efficient alternative for solving elliptic PDEs [Hous88a]. The general formulation of these methods for the discretisation of (1.1)-(1.2) was briefly presented in [Chri89] so we do not include it here. We include though for later reference the formulation of the QSC method for the discretisation of (1.1)-(1.2) in the case that $\alpha \cdot \beta = 0$, ie. the boundary operator is either Dirichlet or Neumann. We assume a uniform rectangular mesh $\Delta = \{(x_i, y_j): i = 0 \text{ to } n \text{ and } j = 0 \text{ to } m\}$ in Ω , on which we define a tensor product of one-dimensional quadratic splines

$$S_{2,\Delta} \equiv S_{2,\Delta} \otimes S_{2,\Delta} \equiv \mathbf{P}_{2,\Delta} \cap C^1(\Omega)$$

with $P_{2,\Delta}$, denoting the space of piecewise biquadratic polynomials with respect to Δ . The one-dimensional quadratic splines are constructed so that the boundary operator equation (1.2) is satisfied exactly at any point on $\partial\Omega$.

We define the set of collocation points T_{Δ} to be the set of midpoints of all subrectangles of Δ . Note that all the collocation points lie in the interior of Ω . We determine the quadratic spline approximation $\nu \in S_{2\Delta}$ to ν in two steps by the following equations:

Step 1:
$$Lv = g \text{ on } T_{\Delta}$$
 (2.1)

Step 2:
$$Lu_{\Delta} = g - P_L v \text{ on } T_{\Delta}$$
 (2.2)

where P_L is appropriate perturbation operator, defined in [Chri88b], [Chri90b]. The first step solution v is a second order approximation to u and u_A is a fourth order one.

The QSC equations (2.1) or (2.2) form a block tridiagonal linear system, of $n \cdot m$ equations. If we assume that the ordering of the collocation points is bottom-up and then left-to-right every block is of order m, the upper and lower bandwidth is m+1 and there are n blocks on the diagonal. Figure 2.1 shows the pattern of non-zero entries in the QSC matrix.

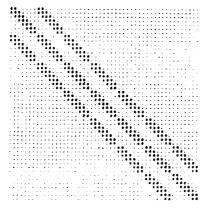


Figure 2.1. Structure of the matrix of QSC equations for n = m = 7. x denotes a non-zero off-diagonal element, d a non-zero diagonal one, while all zero entries are represented by character ".".

3. The Preconditioned Conjugate Gradient (PCG) method for solving linear systems.

In this section our aim is to recall some issues in the parallel implementation of the PCG method. For later reference we include here the steps of the PCG algorithm for solving a linear system Ax = b with preconditioner M, as described in [Golu87]. The superscripts on vectors or scalars denote the iteration number of the algorithm at which the vectors or scalars are computed.

PCG algorithm for
$$Ax = b$$

1. $x^0 = \text{initial guess}$

2. $r^0 = b - Ax^0$
for $k = 1$, maxit

3. if $||r^{k-1}|| \le \epsilon \text{ (or } ||r^{k-1}|| \le ||r^0|| \epsilon \text{) exit}$ else

4. solve $Mz^{k-1} = r^{k-1}$

5. $\beta^k = z^{k-1} r^{k-1} / z^{k-2} r^{k-2}$

6. $p^k = z^{k-1} + \beta^k p^{k-1}$

7. $\alpha^k = z^{k-1} r^{k-1} / p^k Ap^k$

8. $x^k = x^{k-1} + \alpha^k p^k$

9. $r^k = r^{k-1} - \alpha^k Ap^k$

10. endif endfor $x = x^{k-1}$

The computational requirements of every PCG iteration are discussed in detail in [Orte88]. From the above it is clear that the parallel implementation of the PCG method depends very much on the implementation of the linked triad operation (scalar-vector multiplication and vector addition), the inner product operation, the matrix-vector multiplication, the back-and-forward substitutions and the computation of the norm. There are numerous ways to implement the above operations on a parallel machine [Orte88]. They mainly reflect the assignment of the elements of the matrices A and M and the vectors r, z, p and x to the processors.

The CG method without preconditioner follows similar steps, with the exception that M is assumed to be the identity matrix, so the back-and-forward substitutions are avoided. Also, the computation of an inner product can be avoided, when using the Euclidean norm in line 3 of the algorithm.

4. The CG and PCG methods for the QSC equations.

We first experimented with the convergence of the CG iterations applied to the QSC equations. The results show that the number of CG iterations required to satisfy a stopping criterion as in line 3 of the PCG algorithm grows linearly with the square root of the order of the system. In the case where n = m the order of the system is $O(n^2)$, so the number of iterations grows linearly with n. Table (4.1) shows the number of iterations required for the CG-QSC method (CG method applied to QSC equa-

tions) when applied to the problem

$$u_{xx} + u_{yy} = f$$
 in $\Omega = (0,1) \times (0,1)$ (4.1a)

$$u = g \quad \text{in } \partial\Omega.$$
 (4.1b)

f and g are chosen so that the solution to the problem is $u(x,y) = x^{13/2}(x-1)y^{13/2}(y-1)$. The initial guess that was used for step 1 of the QSC method was the zero vector, while in step 2 we use the already computed solution vector from step 1. The relative Euclidean norm of the residual was used for the stopping criterion in line 3 of the PCG algorithm. The desired precision ε was set to 10^{-5} . Figure 4.1 shows graphically the data of Table 4.1.

Table 4.1. Number of iterations required for the convergence of the CG method applied to the QSC equations (2.1)-(2.2) for several grid sizes.

grid size	number of	number of iterations	
n+1	equations	step 1	step 2
5	16	5	5
9	64	11	10
17	256	24	19
25	576	36	26
33	1024	49	34
41	1600	63	42
49	2304	77	50
57	3136	91	58
65	4096	105	66

It is interesting to note that the number of iterations required for convergence of step 2 is exactly n+1 (or n+2) while the slope of the number of iterations curve required for convergence of step 1 is about 1.6875.

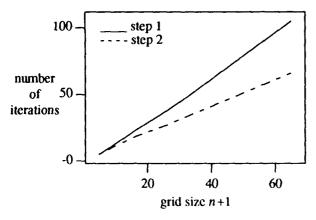


Figure 4.1. Plot of the number of iterations required for the convergence of the CG method when applied to QSC equations for Problem 4.1 versus the grid size in one dimension, for both steps of the QSC method.

In an attempt to explain why the number of iterations required for the convergence of the CG-QSC

method grows proportionally with n, we consider a Helmholtz problem (b = d = e = 0) with constant coefficients (a, c, f constants) and Dirichlet boundary conditions $(\beta = 0)$ and state a theorem which is proved in [Chri90b].

Theorem. Under the assumptions that a, c > 0 and $\pi^2(\frac{a}{(bx-ax)^2} + \frac{c}{(by-ay)^2}) > f$, the spectral norm of the inverse of the matrix of QSC equations in the case of a Helmholtz problem with constant coefficients and Dirichlet boundary conditions is bounded, as $n \to \infty$, $m \to \infty$.

A similar theorem holds in the case of Neumann boundary conditions. Taking in account that the norm of the matrix of QSC equations grows proportionally with n^2 , we conclude that the condition number of the matrix of QSC equations also grows proportionally with n^2 . For a symmetric positive definite system Ax = b we know [Axel84] that the number of CG iterations required for convergence grows proportionally with the square root of the condition number of A. For the case of a Helmholtz problem with constant coefficients and Dirichlet boundary conditions the matrix of QSC equations is symmetric and positive definite, so the number of iterations required for the convergence of the CG-QSC method grows proportionally with n. Figure 4.2 shows the behaviour of the residual of the QSC system as the CG iterations proceed. It is interesting to note that for PDE problems other than the Helmholtz problem, for which the QSC equations are not symmetric we have successfully applied the CG method and its asymptotic behaviour was not extremely different from the one for Problem 4.1.

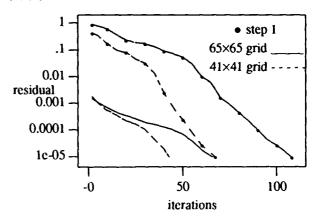


Figure 4.2. Plot of the residual of the QSC system versus the iteration number of the CG algorithm for Problem 4.1, for several grid sizes and for both steps of the QSC method. The residual is in log scale.

We also experimented with the performance of the CG-QSC method as compared with solving the QSC

equations with standard banded LU factorisation (Band-LU). Figure 4.3 shows graphically the results. The slope of the curve of the solution time versus the grid size corresponding to the CG-QSC method is clearly lower than the one for Band-LU. This agrees with the theoretically expected performance of the two methods, since Band-LU is $O(n^4)$, while CG-QSC is $O(n^3)$, where we have again assumed that n = m. Note that this holds for step 1 of the QSC method. For step 2 both Band-LU and CG methods are $O(n^3)$, assuming the factorisation of the matrix of QSC equations is saved from step 1.

The CG method is not the only iterative solver that is faster than direct band solvers for the QSC equations. In [Hous88a] we experiment with several iterative solvers, that outperform the direct ones in both time and memory requirements.

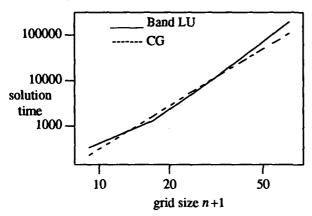


Figure 4.3. Log-log scale plot of the time in milliseconds for the solution of step 1 of the QSC equations with the CG and Band-LU methods versus the grid size in one dimension. The computation was carried out on one processor of the iPSC/2 hypercube.

For matrices of block tridiagonal structure it is quite common to choose as preconditioner the tridiagonal part of the original matrix, in order to accelerate the convergence rate of the CG method. In the case of QSC equations the tridiagonal part T of the matrix is also block-diagonal. When the CG method is applied to the QSC equations arising from the discretisation of a PDE problem with operator other than the Laplace operator, our experiments show that using T as preconditioner accelerated the convergence of the CG method. In [Chri90c] we study the construction of appropriate preconditioners for the QSC equations. In the rest of the paper any reference to the CG-QSC method will assume that an appropriate preconditioner is used whenever necessary.

5. Implementation of the CG-QSC method on hypercube architectures.

In this section we discuss in more detail how the computation involved in the CG-QSC method is mapped on hypercube architectures. Although we limit this discussion in a specific MIMD architecture, most of the ideas presented are straightforward implemented on other type of local memory machines as well as shared memory ones.

5.1. Distribution of the data to processors.

The first thing in the implementation of an algorithm, in which certain parallelism is identified, on a specific local memory machine, is to distribute the data in the local memory of the processors, so that communication is "minimised" and as little as possible data is duplicated. In the case of CG-QSC method the distribution of data is motivated by the parallel implementation of the individual steps of the PCG algorithm as described in Section 3. Although this distribution of data holds only for local memory machines in the case of shared memory machines this distribution reflects the way the processors are going to address the shared memory.

The matrix A of QSC equations is stored by rows in a sparse matrix storage scheme, so that only the non-zero elements of every row are stored. Every processor holds the rows corresponding to one or more blocks in the block notation of the matrix. For simplicity we assume the number of processors P divides n exactly. So every processor will store $\frac{nm}{P}$ equations. Also every processor will store the respective rows of the vectors r, z and x. As far as the direction vector p is concerned a processor will update those components corresponding to the rows of A it stores, but will have storage for the "neighbouring" components, more specifically m positions on the top of the part it is going to update and m positions at the bottom.

5.2. Parallel discretisation of the PDE problem.

The discretisation process of a PDE problem with the collocation methodology is by definition pointwise, so it is totally asynchronous, assuming a distribution of the collocation points to the processors. In our case of QSC with the midpoints as collocation points, and a bottom-up left-to-right numbering of them, the parallel generation of the matrix A can be viewed as a *line collocation* method. Every processor generates the rows of A it is assigned to, that is, the equations corresponding to one or more vertical grid lines, with no need to communicate with any other processor.

5.3. Computing the product of A by a vector.

According to the above assignment of the elements of A and p to the processors, a processor will compute the inner product of the rows of A it holds with the vector p. Due to the block-tridiagonal structure of the matrix A any processor needs to receive at most 2m components of p by other processors, the rest reside already in the local memory of the processor. This fact has two nice effects: First, that only neighbour communication is necessary, if we assume that the assignment of blocks of rows of A to the processors is done according to the standard gray code ordering of the processors. Second, that the amount of data transfer per processor does not grow with the number of processors. It only grows with m, the grid size in one direction. This helps so that the speedup does not degrade much as the number of processors increases.

5.4. Preconditioning.

The assignment of the blocks of T to the processors is similar to that of A. Since T is block diagonal, every processor can work independently for the back-and-forward substitutions of the blocks of T it is assigned to. So preconditioning with T does not increase the communication overhead.

5.5. Computation of inner product of vectors and norms.

For the inner products in lines 5 and 7 of the PCG algorithm the well known fan-in technique is used. More specifically, we use *global fan-in* so that the final result resides in all processors, instead of a fan-in in one processor and a fan-out broadcast of the final result to all other processors. The parallel computation of the norm of the vector depends on the norm used. For the infinity norm a global fan-in comparison scheme is used, while for the Euclidean norm a global fan-in summation.

6. Performance results.

In this section we discuss the performance of the CG-QSC method on various processor configurations of the iPSC/2 hypercube. We refer to the basic computational constructs of the CG-QSC method as discretisation process, solution process and per iteration process. The per iteration process includes the computation of one (P)CG iteration, while the solution process includes the factorisation of the preconditioner (if there is one) and the computation of all iterations.

6.1. Speedup and efficiency.

We first measure the so called *scaled speedup* [Gust88], [Orte88]. According to the definition of scaled speedup we need to choose a different size problem for each processor configuration. For the solution process of

the CG-QSC method, the problem size, that is, the operation counts, is $O(n^3)$, while for each iteration it is $O(n^2)$, as it is for the discretisation process. More specifically, we scale the problem size as follows: Let n be the number of grid points in one dimension, for which we let the CG-QSC program to run on a single processor. We then choose n_P to be such that $n_P^3 = Pn^3$ and let the CG-QSC program to run on P processors for a grid size n_P . Then the scaled speedup for the (solution process of the) CG-QSC algorithm is $\frac{t_1(n)}{t_P(n_P)} \cdot P$, where $t_i(j)$ is the time elapsed for the execution of the program on i processors and grid size j in one dimension. Similarly for the discretisation process and the per iteration computation we choose n_P such that $n_P^2 = Pn^2$. Alternatively, we can compute the scaled speedup for 2 different grid sizes nand n_P as $\frac{t_1(n)}{t_P(n_P)} \cdot \frac{n_P^2}{n^3}$, for the solution process of the CG-QSC algorithm and as $\frac{t_1(n)}{t_P(n_P)} \cdot \frac{n_P^2}{n^2}$, for the discretisation and per iteration processes

In Figure 6.1 we plot the estimated scaled speedup for the discretisation, solution and per iteration processes of the CG-QSC algorithm. The grid sizes for this plot vary from 25 for a single processor to 97 for 32 processors. The discretisation process does not suffer from any communication overhead and the slight degradation of the speedup away from the linear one is due to duplicate computations done in all processors, in order to initialise certain parameters of the problem, as well as to a few differences in the code for a single processor from that for multiple processors. The speedup curves for the solution and per iteration processes look very similar. The degradation of speedup in these cases is mainly due to communication overhead as well as synchronisation and load balancing. We would like to point out that the CG-QSC algorithm is perfectly load balanced as far as computation is concerned. Communication is also well load balanced with the exception of the nearest neighbour communication, that is required for the computation of the matrixvector product Ap, in which the first and last processors remain idle, during the time the others exchange m components of a vector. Also, the unreliability of the

Based on the speedups plotted in Figure 6.1 the efficiency of the discretisation process ranges from 90% to 98%, while the efficiency of the solution and per iteration processes range from 79% to 93%.

hardware might cause some load imbalance.

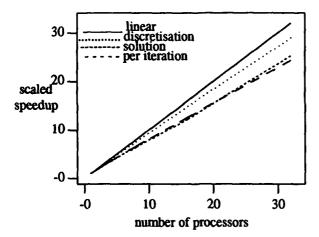


Figure 6.1. Measured speedup for the discretisation, solution and per iteration times of CG-QSC on the iPSC/2 for up to 32 processors configurations.

We next plot the *fixed speedup* for grid sizes 97 and 65 in Figures 6.2 and 6.3 respectively. This turns out to be better than the scaled one for small number of processors, but degrades faster for large number of processors. This comes from the fact that the fixed speedup suffers from the overhead of carrying out small amount of computation in each processor. It is clear that the slope of the fixed speedup curve for large number of processors is lower than that of the scaled one, and that the 65 grid size speedup is worse than the 97 grid size one.

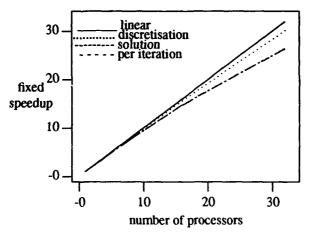


Figure 6.2. Measured speedup for the discretisation, solution and per iteration times of CG-QSC on the iPSC/2 for up to 32 processors configurations and fixed 97×97 grid.

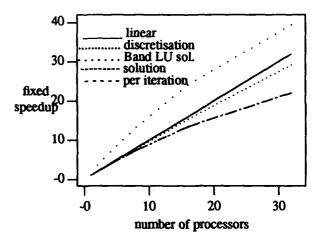


Figure 6.3. Measured speedup for the discretisation, solution and per iteration times of CG-QSC on the iPSC/2 for up to 32 processors configurations and fixed 65×65 grid.

It is interesting to note that the efficiencies of the processors based on the fixed speedups are 100% for 2 processors for both the grid sizes shown in the two figures. More specifically, the efficiency based on fixed speedup varies from 94% (91%) to 100% for the discretisation process and grid size 97 (65), and from 83% (69%) to 100% for the solution and per iteration processes for the same grid size(s).

In Figure 6.3 we also plot the "speedup" of the CG-QSC solution process with respect to the Band-LU solution process carried out in a single processor. This is clearly superlinear, due to the merits of the CG-QSC method. We were unable to run the Band-LU algorithm for larger than 65 grid sizes, due to the limit in the local memory of a processor.

Finally, in Table 6.1 we include some of the numerical data that was used to draw Figures 6.1, 6.2 and 6.3. This table has also a column for the memory requirements of the two methods/solvers. The memory requirements of the CG-QSC method are far less than those for Band-LU. The memory requirements of CG-QSC decrease almost linearly with the number of processors.

Table 6.1. Time in milliseconds on the iPSC/2 for the discretisation, solution and iteration processes of the Band LU factorisation and CG-QSC algorithms for various grid sizes and processor configurations. A "-" means not applicable. The memory requirements are in floating-point numbers.

method	n+1	discret.	solution	per iteration	memory	P
Band-LU	25	8460	5048	-	35712	1
CG-QSC]	8446	5716	141.30	10944	1
		4234	2958	73.10	5496	2
		2156	1664	41.20	2784	4
		1120	1028	25.50	1416	8
Band-LU	33	14952	14228	-	79872	1
CG-QSC		14948	13762	252.70	19456	1
		7492	7040	129.26	9760	2
		3792	3827	70.27	4930	4
	[1952	2222	40.86	2498	8
}	ł	1020	1452	26.74	1282	16
,	,	560	1130	20.78	674	32
Band-LU	41	23268	32960	-	150400	1
CG-QSC	1	5892	7164	106.27	7680	4
Band-LU	49	33360	65464	-	253440	1
CG-QSC		4296	6617	81.25	5568	8
Band-LU	65	59028	197712	-	581632	1
CG-QSC		59020	110876	1022.37	77824	1
		29560	55748	514.06	38976	2
		14876	28425	262.12	19584	4
ł i	ł	7556	14926	137.65	9856	8
		3848	8210	75.76	4992	16
'	1	2024	5022	46.35	2560	32
1	97	132504	349274	2306.25	175104	1
]	}	66432	175340	1157.76	87648	2
		33292	88472	584.16	43968	4
	İ	16820	45314	299.22	22080	8
}	}	8504	23624	156.00	11136	16
	<u> </u>	4400	13217	87.29	5664	32

6.2. Communication time.

As explained before, the communication overhead of the CG-QSC algorithm is due to the matrix-vector multiplication (neighbour communication) and the inner product and norm computation (global communication). In order to verify the theoretically obtained result of Section 5.3, that the communication overhead for computing the product of A by a vector does not increase with the number of processors, we attempt to measure the time spent in communication, during the computation of the product of A by a vector in the following way. We let our code run, skipping all the computation statements and executing only the send/receive operations of the iPSC/2 hypercube. For a better accuracy we let it earry out

several iterations and then take the average of the time elapsed. It is our understanding that in this way we include in our measurements the computation time required for addressing the message buffers and the overhead spent in synchronisation and load balancing. Table 6.2 lists the average communication time measured in this way, for several problem sizes and number of processors.

Table 6.2. Communication time in milliseconds on the iPSC/2 during the computation of the matrix-vector multiplication for various grid sizes and processors configurations.

P $n+1$	2	4	8	16	32
25	0.64	1.28	1.28	1.28	1.28
49	1.28	2.56	2.88	2.88	2.88
81	1.28	2.88	2.88	2.88	2.88

From the results of Table 6.2 it is clear that the communication overhead of our implementation is not affected by the number of processors, except in the case of 2 processors, in which there is only one-way communication. We find these timings quite consistent with our theoretical statements taking in account the factors of clock accuracy and unreliability of the hardware. We also note that the communication overhead is affected (not necessarily linearly) by the size of the problem. This agrees quite well with the communication performance report for the iPSC/2 hypercube [Inte88], where it is stated that the time for node-to-node communication is about the same for 0-100 bytes messages (up to 25×25 grid), it is about double for a message of 104 bytes length, than for one of 100 bytes length, and varies slightly for messages of 104-1024 bytes length (this includes the largest grid size, for which the CG-QSC method was tested).

We have carried out similar experiments in order to measure the time spent in communication during the computation of the inner products and norms in every iteration of the PCG algorithm. Our experiments show that this time increases linearly with the dimension of the hypercube (log(P)), as expected. Based on our experiments the global fan-in summation of the partial inner products takes 2-4 milliseconds for 2-32 processors configurations. Taking in account that every PCG iteration requires 3 times this type of global communication and once the neighbour communication for the matrix-vector multiplication we conclude that the time spent in communication is less than 1% of the total time for the case of 97×97 grid and 2 processors, while it is about 16% of the total time for the same grid size and 32 processors. This means that almost all what is lost in efficiency is due to communication overhead, and it leads us to suggest (once again!) that in order to benefit from the use of a lot of processors, we have to solve problems of appropriately large size.

7. References.

- [Ahlb75] Ahlberg, J. H. and T. Ito, A collocation method for two-point boundary value problems, Math. Comp., 29 (1975), pp. 129-131, 761-776.
- [Arch73] Archer, D. A., Some collocation methods for differential equations, Rice University, Houston, TX, Ph.D. thesis, 1973.
- [Axel84] Axelsson, O. and V. A. Barker, Finite element solution of boundary value problems, Academic Press, 1984.
- [Bjor86] Bjorstand, P. E. and O. B. Widlund, Iterative methods for the solution of elliptic problems on regions partitioned into substructures, SIAM J. Numer. Anal., 23 (1986), pp. 1097-1120.
- [Bram86] Bramble, J. H., J. E. Pasciak and A. H. Schatz, The construction of preconditioners for elliptic problems by substructuring I, Math. Comp., 47 (1986), pp. 103-134.
- [Bram88] Bramley, R. and A. Sameh, A robust parallel solver for block tridiagonal systems, Proceedings of the 1988 International Conference on Supercomputing (ICS88), July 1988, St. Malo, France.
- [Cave72] Cavendish, J. C., A collocation method for elliptic and parabolic boundary value problems, using cubic splines, Univ. of Pittsburgh, PA, Ph.D. thesis, 1972.
- [Chan87] Chan, T. F., Analysis of preconditioners for domain decomposition, SIAM J. Numer. Anal., (1987), pp. 382-390.
- [Chri88a] Christara, C. C., E. N. Houstis and J. R. Rice, A Parallel Spline Collocation Capacitance Method for Elliptic PDEs, Proceedings of the 1988 International Conference on Supercomputing (ICS88), July 1988, St. Malo, France.
- [Chri88b] Christara, C. C., Spline collocation methods, software and architectures for linear elliptic boundary value problems, Ph.D. thesis, Purdue University, IN, U.S.A., 1988.
- [Chri89] Christara, C. C. and E. N. Houstis, A domain decomposition spline collocation method for elliptic partial differential equations, Proceedings of the fourth Conference on Hypercubes, Concurrent Computers and Applications (HCCA4), March 1989, Monterey, CA, U.S.A.
- [Chri90a] Christara, C. C., Schur complement preconditioned conjugate gradient methods for spline collocation equations, to appear in Proceedings of the 1990 International Conference on Supercomputing (ICS90), June 1990, Amsterdam, the Netherlands.
- [Chri90b] Christara, C. C., Quadratic Spline Collocation Methods for Elliptic Partial Differential Equations, Univ. of Toronto, DCS Tech. Rep. (1990), submitted for publication.
- [Chri90c] Christara, C. C., Iterative Solution of Spline Collocation Fquations, Univ. of Toronto, DCS Tech. Rep., in preparation.

- [Conc76] Concus, P., G. H. Golub and D. O'Leary, A generalised Conjugate Gradient method for the numerical solution of elliptic Partial Differential Equations, Sparse Matrix Computations, Bunch J. and D. Rose (eds.), Academic Press, New York 1976, pp. 309-322.
- [Conc85] Concus, P., G. H. Golub and G. Meurant, Block Preconditioning for the Conjugate Gradient method, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 220-252.
- [Dani75] Daniel, J. W. and B. K. Swartz, Extrapolated collocation for two point boundary value problems using cubic splines, J. Inst. Maths Applies, 16 (1975), pp. 161-174.
- [Dryj84] Dryja, M., A finite element capacitance method for elliptic problems on regions partitioned into subregions, Numer. Math., 44, (1984), pp. 153–168.
- [Dryj86] Dryja, M. and W. Proskurowski, Iterative methods in subspaces for solving elliptic problems using domain decomposition, University of Southern California, Tech. Rep. CRI-86-10 (1986).
- [Fyfe68] Fyfe, D. J., The use of cubic splines in the solution of two-point boundary value problems, Comput. J., 17 (1968), pp. 188-192.
- [Golu85] Golub, G. H. and C. F. van Loan, Matrix computations, John Hopkins University Press, 1985 (476 pgs).
- [Gust88] Gustafson, J. L., G. R. Montry and R. E. Benner, Development of Parallel Methods for a 1024-Processor Hypercube, SIAM J. Sci. Statist. Comput., 9, 4 (1988), pp. 609-638.
- [Hage81] Hageman, L. A. and D. M. Young, Applied Iterative Methods, Academic Press 1981.
- [Hest52] Hestenes, M. and E. Stiefel, Methods of Conjugate Gradients for solving linear systems, J. Res. Natl. Bur. Stand. Sect B, 49 (1952), pp. 409-436.
- [Hous87] Houstis, E. N., E. A. Vavalis and J. R. Rice, Convergence of an $O(h^4)$ cubic spline collocation method for elliptic partial differential equations, SIAM J. Numer. Anal., 25 (1988), pp. 54-74.
- [Hous88a] Houstis, E. N., J. R. Rice, C. C. Christara and E. A. Vavalis, Performance of scientific software. Mathematical Aspects of Scientific Software, (J. R. Rice, ed.), Springer Verlag, 1988, pp. 123-156.
- [Hous88b] Houstis, E. N., J. R. Rice and E. A. Vavalis, A Schwartz Splitting variant of Cubic Spline Collocation Methods for Elliptic PDEs, Proceedings of the third Conference on Hypercubes, Concurrent Computers and Applications, January 1988, Pasadena, CA, U.S.A.
- [Inte88] Intel Scientific Computers, iPSC/2 Performance report. Intel Scientific Computers, January 1986, Beaverton, OR, U.S.A.

- [Irod87] Irodotou-Ellina, M., Spline collocation methods for high order elliptic boundary value problems, Aristotle University of Thessaloniki, Greece, Ph.D. thesis, 1987.
- [Irod88] Irodotou-Ellina, M. and E. N. Houstis, An O (h⁶) quintic spline collocation method for fourth-order two-point boundary value problems BIT, 28 (1988), pp. 288-301.
- [Kamm74] Kammener, W. J., G. W. Reddien and R. S. Varga, Quadratic interpolatory splines, Numer. Math., 22 (1974), pp. 241-259.
- [Keye87] Keyes, D. E. and W. D. Gropp, A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s166-s202.
- [Khal82] Khalifa, A. K. and J. C. Eilbeck, Collocation with quadratic and cubic splines, IMA J. Numer. Anal., 2 (1982), pp. 111-121.
- [OLea87] O'Leary, D., Parallel Implementation of Block Conjugate Gradient Algorithm. Parallel Comput., 5 (1987), pp. 127-140.
- [Orte88] Ortega, J. M., Introduction to Parallel and Vector Solution of Linear Systems, Plenum Press, 1988 (305 pgs).
- [Rodr86] Rodrigue, G., Some ideas for decomposing the domain of elliptic Partial Differential Equations in the Schwarz process, Commun. Appl. Numer. Method, 2 (1986), pp. 245-249.
- [Russ72] Russell, R. D. and L. F. Shampine, A collocation method for boundary value problems, Numer. Math., 19 (1972), pp. 1-28.
- [Saka83] Sakai, M. and R. Usmani, Quadratic spline solutions and two-point boundary value problems, Publ. RIMS, Kyoto University, 19 (1983), pp. 7-13.
- [Tang87] Tang, Wei-Pai, Schwartz Splitting and Template Operators, Stanford University, Ph.D thesis, 1987.
- [Varg62] Varga, R. S., Matrix iterative analysis, Prentice Hall, 1962.
- [Youn71] Young, D. M., Iterative Solution of Large Linear Systems, Academic Press 1971.

Acknowledgements

I wish to thank Elias N. Houstis for his suggestions for improving this manuscript. I also wish to thank the Cornell Theory Center for letting me use their iPSC/2 hypercube for free!

Multigrid on Massively Parallel Computers *

David E. Womble Sandia National Laboratories Albuquerque, NM 87185 Brenton C. Young Stanford University Stanford, CA 94305

Abstract

Multigrid is a fast iterative method used to solve linear partial differential equations. However, because the solution of very small problems is inherent in the multigrid iteration, it is difficult to implement efficiently on a massively parallel computer. In this paper, we present an implementation of the multigrid v-cycle that has achieved 84% efficiency on the 1,024 processor NCUBE/ten. We also present a model for the efficiency of multigrid on a parallel computer that depends only on the efficiency of the smoother at each level. This model can be used to verify that it is indeed difficult to obtain extremely high efficiencies (95% to 100%), but that it is relatively easy to obtain moderately high efficiencies (70% to 85%).

Introduction

Multigrid methods are popular iterative method for solving partial differential equations (PDEs) numerically. These methods make use of multiple grids of unknowns to reduce the dependence of the number of iterations required for convergence on the problem size, in contrast to iterative techniques such as Jacobi, Gauss-Seidel and finite precision conjugate gradient iterations. Also, unlike other fast elliptic solvers, multigrid methods are applicable to a wide range of problems, although their implementation becomes more difficult for irregular domains or irregular grids.

Because of its usefulness as an iterative solver for PDEs, there have been many attempts to implement multigrid efficiently on parallel computers [1,3,4,7]. These have generally been carried out for shared

memory computers with 4 to 16 processors and for distributed memory machines with 16 to 256 processors. The best of these implementations have achieved overall efficiencies between 75% and 85% when solving the largest problem possible on their computer. Higher efficiencies are very difficult to attain because of the serial nature of standard multigrid algorithms and the small number of unknowns on coarse grids.

Several variants of the standard multigrid algorithm for parallel computers have also been developed. These include algorithms based on multiple coarse grids [5], algorithms based on simultaneous smoothing on several grids [6], and algorithms for residual splitting to allow the simultaneous reduction of different frequency erorrs [3]. These variants are not always effective as the increased efficiency is offset by increased computational requirements, communication requirements and program complexity.

In this paper, we present an implementation of multigrid for the NCUBE/ten that achieves 84% efficiency when using 1,024 processors. We also present a model of the efficiency of multigrid algorithms that distiguishes between the efficiency of multigrid and the efficiency of the smoother. Finally, we compare our implementation of multigrid on the NCUBE/ten to the predictions of the model.

Implementation

Our multigrid implementation is the v-cycle, in which the iteration begins on the finest grid, progresses sequentially to the coarsest grid, and then returns to the fine grid (Figure 1).

In a parallel implementation of multigrid, processors can be idle on the coarsest levels while the computations proceed at a small number of points. This problem is particluarly severe on massively parallel machines, such as the NCUBE/ten and the Connection Machine. While this problem cannot be eliminated for the v-cycle, its effects can be minimized by

^{*}This work was supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract No. DE-AC04-76DP00789.

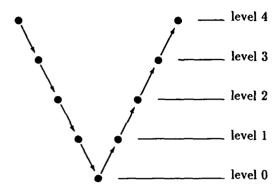


Figure 1. The multigrid v-cycle with five levels

using a process known as agglomeration [7]. Whenever a processor is responsible for a computational domain with too few unknowns, the domain is combined with that of a neighboring processor. If the fine-grid computations are load balanced, many of these combinations can be done in parallel. On a hypercube, this is equivalent to having half of the cube duplicate the computations of the other half. Because each half of the hypercube is also a hypercube, and because the processors that are combining work are connected, nearest neighbor communications are maintained. While agglomeration improves the efficiency at a given level only slightly, it reduces the communication required for the transfer between levels significantly.

The optimum number of unknowns at which agglomeration will occur is a function of the machine architecture. On the NCUBE/ten, we found that two processors should be combined when they are each responsible for two unknowns along any dimension of the problem. The model developed in the next section can be used to verify this and to determine the exact dependence on the hardware parameters.

In our implementation of multigrid, we use a redblack Gauss-Seidel smoother on each level. While this smoother is not as easily parallelized as a Jacobi smoother, the improvement in the multigrid convergence rate is sufficient to justify its use. On the other hand, the use of an SOR smoother results in a comparable convergence rate, but does not parallelize easily.

To transfer information between the grids, we use full weighting restriction and bilinear correction (prolongation) operators [2]. The full weighting re-

striction operator requires one more communication step than injection, which is another commonly used restriction operator. However, because the the full weighting operator is the transpose of the bilinear correction operator, convergence is guaranteed for a wide range of problems.

Numerical Results

The multigrid algorithm described in the previous section was implemented on the NCUBE/ten and tested by solving the following problem:

$$-\frac{\partial^2 u}{\partial x^2} - \epsilon \frac{\partial^2 u}{\partial y^2} = (1 + \epsilon)\pi^2 \sin(\pi x) \sin(\pi y),$$

$$(x, y) \in (0, 1) \times (0, 1),$$

with the boundary conditions

$$u(x,0) = u(x,1) = 0$$
 $x \in [0,1]$
 $u(0,y) = u(1,y) = 0$ $y \in [0,1].$

We discretize this equation by setting $\Delta x = \Delta y = 1/n$, n > 0 and replacing the partial derivatives with second-order finite differences. The mesh is distributed over a $p \times p$ grid of processors by assigning a $(n/p) \times (n/p)$ mesh to each processor. For each run of the program, we set $\epsilon = .1$, smooth once on each level, and consider the iterations to have converged when the relative residual is less than 10^{-9} .

As a measure of the performance of the algorithm, we use efficiency and scaled efficiency. If T(n,p) is the time per iteration for the v-cycle with an $n \times n$ mesh on a $p \times p$ grid of processors, then the efficiency is defined to be

$$e(n,p) = \frac{T(n,1)}{p^2 T(n,p)},$$

which is equivalent to the speedup divided by the number of processors. In our implementation, a 64×64 grid of unknowns is the largest allowed on one processor. Hence, for n > 64 we define the scaled efficiency

$$se(n,p) = \frac{T(n/p,1)}{T(n,p)},$$

which is equivalent to the scaled speedup [8] divided by the number of processors. We note that because the serial run time for one v-cycle is proportional to the number of unknowns, we have $e(n,p) \approx se(n,p)$. Efficiencies and scaled efficiencies for the model problem are shown in Table 1. We see in Table 1 that the degradation in performance due to idle processors as n/p decreases is quite severe. For the cases n=4 and n=8, we have efficiencies less than $1/p^2$, which corresponds to a speedup of less than one. We also note that by scaling the problem with the number of processors, we obtain relatively good efficiencies, even on 1,024 processors. We conclude that the multigrid v-cycle has a relatively high serial content, and that scaling the problem size with the number of processors is more important than with more parallelizable algorithms such as Jacobi relaxation.

Model of Multigrid Efficiency

In this section, we develop an expression for efficiency of the multigrid v-cycle in terms of the efficiency of the smoother used at each level. Although the model is developed only for the v-cycle, the techniques are applicable to any cycle [9].

We suppose that we are approximating the solution to a d-dimensional PDE at n^d evenly spaced mesh points distributed among p^d processors. We use using a k-level, $k \leq \log_2 n$ multigrid algorithm, where level 1 is the coarsest grid, and level k is the finest grid. For $i=1,2,\ldots,k$, we let $e_i(n,p)$, $o_i(n,p)$ and $c_i(n)$ denote the efficiency, the parallel overhead and the computational work for the i-level multigrid algorithm. We note that the computational work $c_i(n)$ does not depend on the number of processors p. Similarly, we let $\Delta e_i(n,p)$, $\Delta o_i(n,p)$ and $\Delta c_i(n)$ denote the corresponding quantities for the smoother at level i. Our goal is to develop an expression for $e_k(n,p)$ in terms of $e_{k-1}(\frac{n}{2},p)$ and $\Delta e_k(n,p)$.

The definition of efficiency is

$$e_i(n,p) = \frac{c_i(n)}{o_i(n,p) + c_i(n)}.$$

Because

$$c_i(n) = c_{i-1}(\frac{n}{2}) + \Delta c_i(n),$$

and

$$o_i(n,p) = o_{i-1}(\frac{n}{2},p) + \Delta o_i(n,p),$$

we can write the efficiency as

$$e_i(n,p) = \frac{1}{1 + \frac{\sigma_{i-1}(\frac{n}{2},p) + \Delta\sigma_i(n,p)}{c_{i-1}(\frac{n}{2}) + \Delta c_i(n)}}.$$

Eliminating $o_{i-1}(n, p)$ and $\Delta o_i(n, p)$ from this equa-

tion yields

$$e_{i}(n,p) = \frac{e_{i-1}\left(\frac{n}{2},p\right)\Delta e_{i}(n,p)\left(1+\frac{\Delta c_{i}(n)}{c_{i-1}\left(\frac{n}{2}\right)}\right)}{\Delta e_{i}(n,p)+\frac{\Delta c_{i}(n)}{c_{i-1}\left(\frac{n}{2}\right)}e_{i-1}\left(\frac{n}{2}\right)}.$$

The ratio $\Delta c_i(n)/c_{i-1}(\frac{n}{2})$ depends only on the computational work in the serial algorithm and can be approximated. The work required by the smoother on level j can be written in terms of the work on level i as follows:

$$\Delta c_j\left(\frac{n}{2^{i-j}}\right) = \left(\frac{1}{2^{d(i-j)}}\right) \Delta c_i(n), \qquad j=1,\ldots,k.$$

Now

$$c_i(n) = \sum_{j=1}^i \Delta c_j \left(2^{j-i} n \right)$$

Thus, for a v-cycle, we have

$$c_{i-1}\left(\frac{n}{2}\right) \approx \Delta c_i(n) \left(\frac{1}{2^d-1}\right)$$
.

Substituting this relationship into (1) yields the recursion

$$e_i(n,p) \approx \frac{2^d}{\frac{1}{e_{i-1}(\frac{n}{2},p)} + (2^d - 1)\frac{1}{\Delta e_i(n,p)}}, \qquad i > 1.$$
(2)

The initial condition,

$$e_1(2^{1-k}n,p) = \Delta e_1(2^{1-k}n,p),$$

where k is the number of levels, simply states that the efficiency of multigrid is the same as the efficiency of the smoother when only one level is used.

To actually predict the efficiencies of a multigrid v-cycle, we need to be able to predict the efficiency of the smoother, $\Delta e_i(n,p)$. For red-black Gauss-Seidel, we use

$$\Delta e_i(n,p) = \frac{C_1 \left(\frac{n}{p}\right)^2}{C_1 \left(\frac{n}{p}\right)^2 + C_2 \left(\frac{n}{p}\right) + C_3 \log_2 p^2 + C_4},$$
(3)

with

$$C_1 = 2.13 \times 10^{-4} \pm 7.2 \times 10^{-8}$$

 $C_2 = 4.23 \times 10^{-4} \pm 4.8 \times 10^{-6}$
 $C_3 = 8.23 \times 10^{-4} \pm 1.2 \times 10^{-5}$
 $C_4 = 9.73 \times 10^{-3} \pm 9.5 \times 10^{-5}$

This equation is based on operation counts and a least squares fit to timing data for the

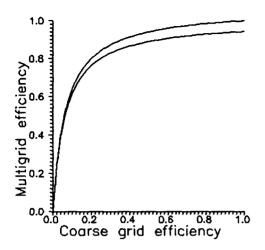


Figure 2. The efficiency of an i-level multigrid v-cycle as a function of the efficiency of an (i-2)-level multigrid v-cycle. If the efficiency of the smoother on the finest level is greater than 95% and on the second finest level is greater than 90%, then the i-level multigrid efficiency must lie between the two curves.

NCUBE/ten [9]. Conbining (2) and (3) yields the predictions of multigrid efficiency shown in Table 2.

We see from the multigrid efficiency model (2) that the coefficient of the fine-grid efficiency is much larger than that of the coarse-grid efficiency. We conclude that the efficiency of the fine-grid smoother influences the efficiency of multigrid more strongly than the coarse-grid smoothing. Thus, if an efficient smoother, such as Jacobi or red-black Gauss-Seidel, is used, moderately high efficiencies for multigrid can be achieved easily. Figure 2 demonstrates this.

We also note from (2) that as the dimension, d, increases, the weighting of the fine-grid efficiency increases. Thus, even though the number of points per processor on coarse grids decreases more rapidly in higher dimensional problems, it should be possible to maintain good efficiencies for multigrid algorithms in higher dimensions.

Finally, we can verify that agglomeration should occur when a processor is responsible for two unknowns along any dimension. In particular, for the two dimensional problem, agglomeration should occur at level *i* when

$$\Delta e_i(n,p) < \frac{1}{4} \Delta e_{i+1}(n,\frac{p}{2}).$$

That is, agglomeration should occur when the loss of efficiency due to communication is more than that due to idle processors. Substituting the efficiency of the red-black Gauss-Seidel smoother (3) into this

inequality yields

$$3C_1\left(\frac{n}{p}\right)^2+C_2\left(\frac{n}{p}\right)-4C_3<0.$$

Solving, we find that agglomeration should occur when $-2.62 < n/p < 1.96 \approx 2$.

Conclusions

Our implementation of the multigrid v-cycle on the 1,024 processors NCUBE/ten achieved 84% efficiency, demonstrating that multigrid algorithms can be implemented on massively parallel computers efficiently. We note that the algorithm chosen for parallelization is one of the most effective serial algorithms for the solution of partial differential equations. It was not chosen because of any inherent parallelism.

We also developed a model for the efficiency of the multigrid v-cycle that depends only on the efficiency of the smoother at each level and the dimension of the problem. Using this model, we showed that with an efficient fine-grid smoother, relatively high efficiencies are relatively easy to obtain. In particular, multigrid efficiencies of 75% to 95% should be attainable on most parallel computers. We also concluded that similar multigrid efficiencies can be attained for higher dimensional problems.

References

- A. Brandt. Multigrid solvers on parallel computers. In M. H. Schultz, editor, Elliptic Problem Solvers, pages 39-84, Academic Press, New York, 1981.
- [2] W. L. Briggs. A Multigrid Tutorial. SIAM, Philadelphia, PA, 1987.
- [3] T. F. Chan and R. S. Tuminaro. Design and implementation of parallel multigrid algorithms. In S. F. McCormick, editor, Multigrid Methods: Theory, Applications, and Supercomputing, pages 101-115, Marcel Dekker, Inc., 1988.
- [4] T. F. Chan and R. S. Tuminaro. A survey of parallel multigrid algorithms. In A. K. Noor, editor, Parallel Computations and Their Impact on Mechanics, AMD-86, pages 155-170, ASME, 1988.
- [5] P. O. Frederickson and O. A. McBryan. Parallel Superconvergent Multigrid. Technical Report CTC87TR12, Cornell Theory Center, 1987.

- [6] D. Gannon and J. van Rosendale. On the structure of parallelism in a highly concurrent PDE solver. J. Par. Dist. Comp., 3:106-135, 1986.
- [7] U. Gärtel. Parallel Multigrid Solver for 3D Anisotropic Elliptic Ploblems. Technical Report 390, Gesellschaft Für Mathematik und Datenverarbeitung, 1989.
- [8] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. SIAM J. Sci. Stat. Comp., 9(4):609-638, 1988.
- [9] D. E. Womble and B. C. Young. A Model and Implementation of Multigrid for Massively Parallel Communities. Technical Report SAND89-2781, Sandia National Laboratories, 1990.

Table 1. Efficiency and and scaled efficiency for the multigrid v-cycle on the NCUBE/ten when solving an n x n problem on a p x p grid of processors. Dashes (—) correspond to cases for which no timing data exists. Efficiencies appear below the line in the table; scaled efficiencies appear above the line.

	$p \times p$					
$n \times n$	1 × 1	2×2	4 × 4	8 × 8	16 × 16	32×32
4 × 4	1.00	0.125			_	
8 × 8	1.00	0.215	0.047		_	_
16 × 16	1.00	0.392	0.104	0.027		-
32×32	1.00	0.630	0.244	0.075	0.019	
64×64	1.00	0.818	0.449	0.200	0.059	0.015
128 × 128		0.915	0.725	0.434	0.169	0.049
256 × 256	_		0.869	0.694	0.395	0.146
512 × 512	_	_		0.858	0.668	0.362
1024 × 1024	_			_	0.844	0.642
2048×2048	L <u> —</u> .					0.835

Table 2. Predicted efficiencies for the multigrid v-cycle solving an $n \times n$ problem on a $p \times p$ grid of processors using the red-black Gauss-Seidel smoother.

	$p \times p$					
$n \times n$	1 × 1	2×2	4 × 4	8 × 8	16 × 16	32×32
8 × 8	1.00	0.235	0.049	0.012	0.003	0.001
16×16	1.00	0.433	0.110	0.029	.0.007	0.002
32×32	1.00	0.667	0.254	0.077	0.020	0.005
64×64	1.00	0.840	0.499	0.200	0.057	0.015
128×128	1.00	0.929	0.745	0.435	0.162	0.045
256×256	1.00	0.969	0.893	0.704	0.383	0.134
512×512	1.00	0.986	0.957	0.876	0.664	0.339
1024×1024	1.00	0.994	0.983	0.952	0.859	0.627
2048×2048	1.00	0.997	0.993	0.981	0.947	0.842

A Parallel Algorithm for Solving Higher KdV Equations on a Hypercube

Thiab R. Taha
Department of Computer Science
The University of Georgia
Athens, Georgia 30602

Abstract

Taha and Ablowitz derived numerical schemes by methods related to the inverse scattering transform (IST) for physically important equations such as the Korteweg-de Vries (KdV) and modified Korteweg-de Vries (MKdV) equations. Experiments have shown that the IST numerical schemes compare very favorably with other numerical methods. In this paper an accurate numerical scheme based on the IST is used to solve non-integrable higher KdV equations, for instance:

$$u_r + u^4 u_x + u_{xxx} = 0$$

It has been conjectured that the above equation admits a self-focusing singularity. The proposed numerical scheme is used to investigate this phenomenon. The implementation of the IST scheme leads to a huge periodic banded system of equations to be solved at each time step, which requires a large amount of computing time if a serial computer is used. A vector and parallel implementation of the proposed scheme on an Intel iPSC/2 hypercube is carried out, and the numerical results are discussed.

1. Introduction

It has been shown that the higher nonlinear Schrödinger (NLS) equation

$$iq_t + q_{xx} + A |q|^{2p} q = 0, p \ge 2$$
 (1)

under certain conditions admits a self-focusing singularity [1], which means that the solution of Eq. (1) blows up in finite time. This suggests that the higher nonlinear KdV equation

$$u_t + A u^p u_x + u_{xxx} = 0, p > 3$$
 (2)

has a self-focusing singularity [2].

Recently, there has been a lot of theoretical and numerical research in order to investigate this phenomenon (see Bona et al. [3], and the references there in). Numerical simulations of solutions of Eq. (2) (see Fornberg & Whitham [4], Bona et al [5]) confirm that its solitary-wave solutions are unstable if $p \ge 4$, and in fact, that neighbouring solutions emanating from smooth initial data appear to form singularities in finite time. This paper deals with a numerical investigation of the blow-up for the higher KdV equation

$$u_t + u^4 u_x + u_{xxx} = 0 (3)$$

using an accurate numerical scheme based on the IST. The proposed numerical scheme is based on an IST numerical scheme derived by Taha and Ablowitz for the KdV and MKdV equations. Experiments have shown that the IST numerical schemes compare very favorably with other numerical methods [6,7].

In order for the singularity to be properly resolved, the mesh sizes in the directions of x and t have to be taken very small. Therefore the implementation of the proposed numerical scheme on a serial computer requires a large amount of computing time.

In this paper a parallel algorithm for the above scheme is designed and implemented on an Intel iPSC/2 hypercube, and the numerical results are discussed.

2. The proposed numerical scheme

The proposed numerical scheme which is based on the IST for Eq. (3) is [6]

$$\frac{u_n^{m+1}-u_n^m}{\Delta t}=\frac{1}{2(\Delta x)^3}\left[u_{n-1}^{m+1}-3\,u_n^{m+1}\right]$$

$$+ 3 u_{n+1}^{m+1} - u_{n+2}^{m+1} + u_{n-2}^{m} - 3 u_{n-1}^{m}$$

$$+ 3 u_{n}^{m} - u_{n+1}^{m}] - \frac{1}{4 \Delta x} [(u_{n}^{m})^{2}$$

$$- (u_{n}^{m+1})^{2} + \frac{1}{3} \{u_{n+1}^{m+1} (u_{n}^{m+1})^{2} + u_{n+1}^{m+1} + u_{n+2}^{m+1}) - u_{n-1}^{m} (u_{n}^{m})^{2}$$

$$+ u_{n-1}^{m} + u_{n-2}^{m} \} [(\frac{u_{n}^{m} + u_{n}^{m+1}}{2})^{3} (4)]$$

The truncation error of this scheme is $O((\Delta t)^2) + O((\Delta x)^2)$. This scheme is applied to Eq. (3) subject to a Gaussian profile of the form

$$u(x,0) = \eta e^{-\left(\frac{x}{\gamma}\right)^2}, \qquad (5)$$

with $\eta = 3$, and $\gamma = 8$ as an initial condition, and periodic boundary conditions on the interval [-40, 40] are imposed.

3. A parallel implementation of the proposed scheme.

Eq. (4) can be written as

$$-u_{n-1}^{m+1} + (3+\varepsilon)u_n^{m+1} - 3u_{n+1}^{m+1} + u_{n+2}^{m+1} = B_n,$$
 (6)

where

$$\varepsilon = \frac{2(\Delta x)^3}{\Delta t} .$$

and

$$B_{n} = -u_{n+1}^{m} + (3 + \varepsilon)u_{n}^{m} - 3u_{n-1}^{m} + u_{n-2}^{m}$$

$$- \frac{1}{2}(\Delta x)^{2}[(u_{n}^{m})^{2} - (u_{n}^{m+1})^{2}$$

$$+ \frac{1}{3}\{u_{n+1}^{m+1}(u_{n}^{m+1} + u_{n+1}^{m+1} + u_{n+2}^{m+1})$$

$$- u_{n-1}^{m}(u_{n}^{m} + u_{n-1}^{m} + u_{n-2}^{m})\}](\frac{u_{n}^{m} + u_{n}^{m+1}}{2})^{3} (7)$$

One way to implement this scheme is to solve a periodic banded system of equations at each time step:

$$=\begin{bmatrix} B_{-N} \\ B_{-N+1} \\ B_{-N+2} \\ \vdots \\ \vdots \\ B_{N-3} \\ B_{N-2} \\ B_{N-1} \end{bmatrix}$$
(8)

where $\alpha = 3 + \epsilon$. The above system can be solved on a hypercube by using a modified version of an efficient parallel algorithm for banded systems [8,9].

Another way to implement the scheme on the iPSC/2 system is to use the sweeping/iteration technique presented in [6]. To explain the sweeping/iteration technique, we seek an equation of the form

$$u_{n+1}^{m+1} = au_n^{m+1} + b_n^{m+1} (9)$$

which is suitable for computing u_n^{m+1} explicitly by sweeping to the right. For stability $|a| \le 1$. Repeated substitution of Eq. (9) into Eq. (6) to eliminate u_{n+2}^{m+1} , u_{n+1}^{m+1} , and u_n^{m+1} in favor of u_{n-1}^{m+1} gives

$$b_{n+1}^{m+1} + (a-3)b_n^{m+1} + (a^2 - 3a + 3 + \varepsilon)b_{n-1}^{m+1}$$

$$+ (a^3 - 3a^2 + 3a - \varepsilon a - 1)u_{n-1}^{m+1}$$

$$= B_n$$
(10)

Requiring the u_{n-1}^{m+1} term to drop out determines a (uniquely since $|a| \le 1$) as a solution of

$$(a-1)^3 + \varepsilon a = 0 \tag{11}$$

and leaves for b_n (at the new time step) a second order difference equation given by

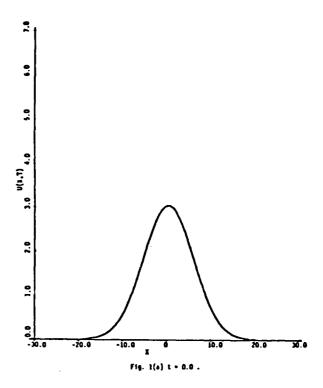
$$b_{n-1} = (3a - a^2)b_n - ab_{n+1} + aB_n$$
 (12)

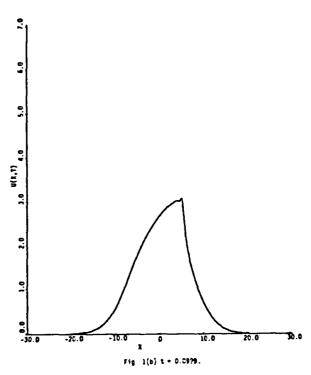
(where Eq. (12) is obtained from Eq.'s (10) and (11)), which is suitable for computing the b's explicitly by sweeping to the left. This method is well suited for serial computers but not for pipeline or vector systems due to the recursive nature of Eq.'s (9) and (12). To implement the scheme on the extension board of an intel iPSC/2 hypercube the cyclic reduction method [10] is used to solve the bidiagonal linear system with a non zero element on the upper right hand corner generated from Eq. (9). On the other hand, the cyclic reduction method for the periodic tridiagonal system generated from Eq. (12) proved to be unstable. An efficient vector algorithm such as a modified LU decomposition for tridiagonal systems with partial pivoting is suggested. It is to be noted that the rest of the computations of the sweeping technique are well suited for vector operations. To implement the sweeping technique on a parallel system such as the hypercube, the systems generated from Eq.'s (9) and (12) should be solved by modified parallel algorithms for bidiagonal and tridiagonal systems respectively [8,9].

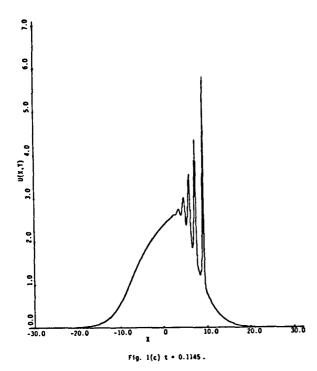
4. Numerical Experiments

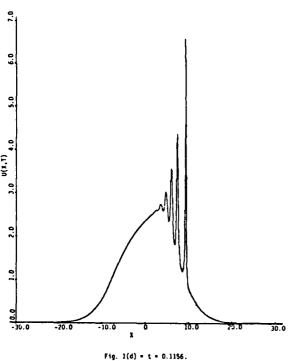
numerical proposed scheme implemented on the iPSC/2 system. The system given in Eq. (8) is solved by an iterative SOR parallel algorithm, and it is found that this method does not converge. Then the proposed scheme is implemented on the extension vector board of an intel iPSC/2 system by using a cyclic reduction method for Eq. (9), properly vectorizing Eq. (7) in order to calculate the B's (otherwise it will not vectorize properly and it will give wrong results), and leaving Eq. (12) unvectorized. preliminary experiments indicate that the above algorithm is four times faster than its serial version. It is to be noted that more work has to be done in order to vectorize Eq. (12). Also, more work has to be done in order to parallelize the sweeping algorithm. According preliminary experiments the solution of the higher KdV equation (3) blows up at t = 0.1157 (see Fig. 1).

Figure 1. Displays the evolution under Eq. (3) of a Gaussian profile given in Eq. (5) as an initial condition on the interval [-40, 40]. $\Delta x = 0.0391$ and $\Delta t = 0.0001$. (a) t = 0.0, (b) t = 0.0999, (c) t = 0.1145, (d) t = 0.1156.









Acknowledgements:

The author would like to thank E. R. Canfield for his careful reading of this paper.

This research has been supported in part by the U. S. Army Research Office and the National Science Foundation Grants No. CCR-8717033 and No. CDA-8920953.

References

- [1] Zakharor, V.E. & Synakh, V.S. (1976) The nature of the self-focusing singularity, Sov. Phys. JETP., 41, pp. 465-468.
- [2] Ablowitz, M. & Segur, H., (1981) Solitons and the inverse scattering transform, SIAM, Philadelphia.
- [3] Bona, J.L., Souganidis, P.E., & Strauss, W.A. (1987) Stability and instability of solitary waves of Korteweg-de Vries type, Proc. R. Soc. Lond. A 411, pp. 395-412.
- [4] Fornberg, B. & Whitham, G. (1978) A numerical and theoretical study of certain nonlinear wave phenomena, Phil. Trans. R. Soc. Lond. A 289, pp. 373-404.
- [5] Bona, J.L. Dougalis, V.A. & Karakashiam, O.A. (1986) Fully discrete Galerkin methods for the Korteweg-de Vries equation, Computat. Math. Appl. 12A, pp. 859-884.
- [6] Taha, T.R., & Ablowitz, M. (1984) Analytical and Numerical Aspects of Certain Nonlinear Evolution Equations. III. Numerical, Korteweg-de Vries Equation, J. Comput. Phys. 55, 2, pp. 231-253.
- [7] Taha, T.R. & Ablowitz, M.J. (1988) Analytical and Numerical Aspects of Certain Nonlinear Evolution Equations. IV. Numerical, Modified Korteweg-de Vries Equation, J. Comput. Phys. 77, 2, pp. 540-548.
- [8] Gallivan, K.A., Plemmons, R.J. & Sameh, A.H. (1990) Parallel algorithms for dense linear algebra computations, SIAM Rev., 32, 1, pp. 54-135.
- [9] Ortega, J. (1988) Introduction to Parallel and Vector Solution of Linear Systems, Plenum, New York.
- [10] Taha, T.R. (1990) Solution of Periodic Tridiagonal Linear systems on a Hypercube, in the Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, SC, to appear.

The Triangle Method for Saving Startup Time in Parallel Computers

(preliminary version)

Horst Eissfeller Silvia Melitta Müller

Computer Science Department
University of Saarland
D-6600 Saarbrücken 11, F.R.G.

Abstract:

We present a new parallel implementation of explicit time stepping methods for time dependent equations in one or two spatial dimensions. The aim is to minimize the number of data transfers, to get faster algorithms. In one spatial dimension, t explicit time steps on p processors using a grid of size n need O(t n/p) arithmetical operations and O(τ) startup operations. The triangle method also requires Out n/p arithmetical operations but only O(τ p/n / startup operations. In two spatial dimensions, using a grid of size n n and given the same algorithm, the startup time of $O(\tau)$ operations using the conventional approach is considerably reduced to O(t √p/n) startup operations. All constants regarding the O notation are less than 5.

Introduction:

The efficiency of parallel numerical algorithms should be maximal (efficiency a sequential

+ Research partially funded by DFG, SFB 124

runtime / parallel runtime + number of processors). Therefore the amount of communication should be small. Communication consists of the startup time and the time to transfer data. The startup time is the time to build up a connection between the processors. For real parallel computers this time is very large, due to software protocol. The following examples illustrate this:

PARSYTEC Super Cluster: 750

INTEL iPSC 2 : 2000 SUPRENUM 1 : 3000

(startup time measured in multiples of the time needed for 1 floating point operation; SUPRENUM supports asynchronous transfer of data, which is more general and harder to implement than sychronous transfer).

As a model of computation we use p processors interconnected by a crossbar. A hypercube or a reconfigurable two dimensional mesh would also suffices. One floating point operation has cost A. A transfer of n numbers costs S+n/B, where S stands for startup and B for bandwidth. All other operations have cost O.

Implementation for one spatial dimension:

Solving the following initial boundary value problem (compare [1])

$$u_{t}(t, x) = \sigma u_{xx}(t, x)$$

 $t > 0$, $x \in I \equiv [0, 1]$
 $u(0, x) = \phi(x)$, $x \in I$
 $u(t, x) = 0$, $t > 0$, $x \in \partial I \equiv \{0, 1\}$

one often gets explicit iterative formulas like

$$u_{i+1 k} = u_{i k} + \Delta t \sigma / \Delta x \left\{ u_{i k+1} - 2 u_{i k} + u_{i k-1} \right\}$$

 $i \ge 0, \ 0 \ k \ n, \ n=1/\Delta x$
 $u_{0 k} = \phi(k \Delta x), \quad u_{i 0} = u_{i n} = 0$.

The goal is the computation of $u_{i\,k}$, $1=i=\tau$, 0=k-n. This will be done by explicit time stepping as the above formula suggests. During one such step one computes the values $u_{i+1\,k}$, 0=k-n, using the results of the last iteration $u_{i\,k}$, 0=k+n.

Standard method:

Initially one distributes the gridpoints over the processors (0..p-1). During a time step processor x sends $u_{i-nx/p}$ to processor x-1 and $u_{i-n(x+1)/p-1}$ to processor x+1. After that it computes the values $u_{i+1,r}$, nx/p-r-n(x+1)/p 1. Thus τ time steps have the cost: $\tau(4n/pA+2S+2/B)$.

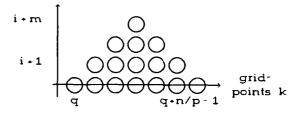
Triangle method:

Starting with the same distribution the τ iterations are computed in blocks of $\lfloor n/(2p) \rfloor$ iterations, without changing the data dependence. The computation of one block consists of three steps. Suppose that n/p = 2m + 1 and processor x updates the values u_i ,

q = k + q + n/p.

During the <u>first step</u> processor x computes $u_{i+j|k}$, $1 \le j \le m$, $q+j \le k \le q+n/p-j$. This can also be seen as building up a triangle over the processors domain.

iterations i

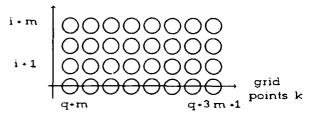


Values computed during the first step (processor x)

During the <u>second</u> step data are exchanged in the following way. Processor x sends the values $u_{i,j,k}$, 1-j-m-1, q+j-k-q+j+1 and $u_{i,q+1}$, $u_{i,m,q,m}$ to processor x-1. During the third step they complete the last

During the <u>third</u> step they complete the last m time steps: processor x computes the values $u_{i + j \mid k}$, 1 = j = m, q + n/p + j = k = q + n/p + j.

iterations i



- O values computed during the third step (processor x)
- O values computed during the first step

For the next m steps processor x updates the values $u_{i+m,q+m}$. $u_{i+m,q+3m}$. Thus τ iterations cost:

$$t4n/pA+[2tp/n](S+n/(pB))$$

The amount of data transferred and of arithmetic performed remains the same, but

the number of startup operations is considerabely reduced. For example, for p = 32, S = 750, $n = \tau = 10^4$, A = 1, B = 1/28 one gets the efficiencies:

eff
$$_{\infty nv}$$
 = 44.55%, eff $_{triangle}$ = 95,4%

Implementation for two spatial dimensions:

Now we consider the two dimensional problem compare [1] :

$$\begin{array}{c} u_{+}^{-}(t,x,y)=-\sigma\bigtriangleup u^{+}(t,x,y)\\ & t>0\;,\;\; (x,y)=\Omega\cap\mathbb{R}^{2}\;,\\ & \Omega\cap\left[\;0,1\;\right]\cdot\left[\;0,1\;\right]\\ u+0,\;x,\;y^{-}=\phi\cdot x,\;y^{-}-x,\;y^{-}+\Omega\\ u+t,\;x,\;y^{-}=\vartheta\cdot x,\;y^{-}-t>0\;,\;\; (x,y)=\omega\Omega. \end{array}$$

Using step sizes $\triangle x$, $\triangle y$ and $\triangle t$, the usual 5 point star to approximate $\triangle u$ and the forward differential quotient to approximate u_{*} , one gets :

To update a point one needs the values of its 4 direct neighbours as well as its own.

Standard method:

Conventionally, to assign processors, one covers the area with squares or bands; if the

area is a rectangle with length n and width $m + (m \ll n)$ it is better to use bands. During each step the processors exchange their borders and update their whole area. For t iterations one gets:

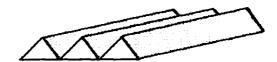
using squares:

$$\tau (6n^2/p A + 4S + 4n/(\sqrt{p} B))$$
 (1) using bands:

$$\tau (6nm/p A + 2S + 2m/B)$$
 (2)

Triangle method:

Dividing the area into bands (m «n), it is easy to reduce our method to the one dimensional case. During each round of computation the processors build up a prism over their domain. They exchange their borders and during the next round they complete the last iterations and build up new prisms and so on.



plane, transfered to the neighbour



prism of the last computation step

Now r iterations cost only:

Comparing this with (2), the number of startup operations is reduced without increasing other operations.

If m n it is better to use squares, but then the method is more difficult. One needs 3 rounds of computation and 4 of communication to compute $n/2\sqrt{p}$ time steps. First each processor builds up a pyramid over its domain. The height of the pyramid is $n/2\sqrt{p}$ time steps. We refer to fig. 1. To carry on, the borders must be exchanged. Each processor sends the two upper levels of face F1 of its pyramid to the neighbour in front of it and the two upper levels of face F2 to its right neighbour. Thus one needs two transfers with

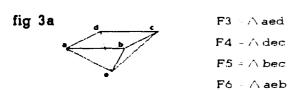
$$2\sum_{i=0}^{\lfloor n/2\sqrt{p}\rfloor-1} (2i+1) = n^2/2p$$

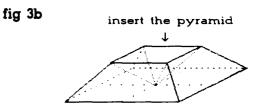
data per transfer. After the communication each processor completes the last $n/2\sqrt{p}$ iterations as far as possible. Then the fig. 2 arises locally:





Only the pyramid shown in fig. 3 remains to be computed. The planes F5 and F6 are in the same processor. Thus only two planes must be transferred. This can be done by two transfers with $n^2/2p$ data each.





After the transfer the remaining pyramid can be computed. For τ iterations one needs: $\tau 6n^2/p A + 2\tau \sqrt{p}/n (4S + 2n^2/(pB))$.

The number of startups is much smaller than in (1). The number of all other operations remains unchanged. For example, for p = 32, S = 750, $\tau = n = m = 10^2$, A = 1, B = 1/28 one gets the efficiencies:

The new implementation has one disadvantage in the two dimensional case. Because different iterations are computed together, more data have to be stored at the same time. The conventional implementation gets problematic with regard to the efficiency only for small amounts of data per processor ($n^2/p = 10^3$). In this case our method is more efficient and practicable. With a trick it is possible to use only 4.5 times more memory than the standard method and a little extra administration. The pyramids are discrete ones, which are stored in planes. From plane to plane they lose one ring. With respect to the algorithm only the two upper levels of each face of the pyramid are necessary, to continue the computation. Therefore it is enough to store the two most outside rings of each plane. Allocating two matrices of size 3n/2p + 3n/2pper processor all data can be stored, as we show now. The planes with even numbers are stored in the first matrix and the others in

the second one. During the first round of computation each processor works on its original domain (fig. 4).

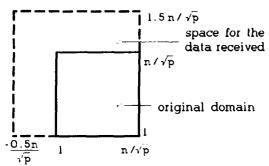


fig. 4: dimension of the matrix

The data received during transfer are stored in a simular manner in the remaining parts of both matrices (fig. 5).

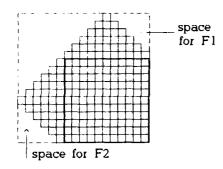


fig. 5 a : data received during the first transfer

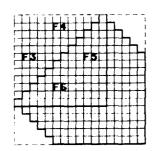


fig. 5 b: data received during the second transfer

After n/2 \sqrt{p} iterations are computed, the processor works over a different part of the matrices (fig. 6).

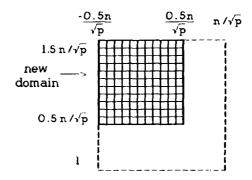
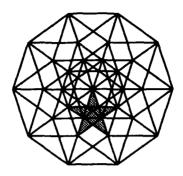


fig. 6: the original domain for the next iterations

During the next block of iterations the processor transfers its borders in the other direction. Therefore all iteration can be computed on the two matrices. In the convential implementation n^2/p space is necessary to store the domain of a processor and $4 \, n/\sqrt{p}$ space to store the borders exchanged. The Triangle Method needs $4.5 \, n^2/p$ space per processor, that is no more than $4.5 \, r^2/p$ space.

References:

[1] Todd, J. (1962) Survey of Numerical Analysis, Mc Craw Hill Book Company, INC, New York, pp. 419.



The Fifth Distributed Memory Computing Conference

22: Concurrent Simulation Paradigms

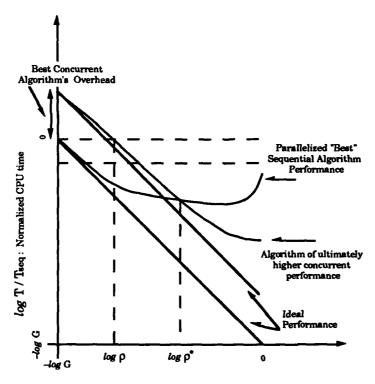
A MINI-SYMPOSIUM ORGANIZED BY:
Anthony Skjellum and Manfred Morari
California Institute of Technology
and
Sven Mattisson and Lena Peterson
Lund Institute of Technology

Concurrent Simulation Paradigms Mini-Symposium

The Concurrent Simulation Paradigms Minisymposium addressed the use of distributed memory computers in the solution of large-scale systems of ordinary differential and differential-algebraic equations (ODE's and DAE's). The solution of large-scale systems of parabolic partial differential equations is also considered in the paper by Vandewalle. Applications from electrical and chemical engineering are presented here, specifically the transient response of VLSI circuits and a dynamic flowsheet simulation used on networks of distillation columns.

The key characteristics shared by these problems are their large-scale and inhomogeneous nature, sparse connectivity, stiffness, and widely varying timescales. It is not meaningful to "scale" these problems. Traditional numerical methods are considered in two of the papers. The Concurrent DASSL and ESACAP efforts utilize parallelized sequential numerical analysis. A novel numerical method, Waveform Relaxation, is considered by four papers and realized, for instance, in the CONCISE VLSI circuit simulator. Speedups achievable on medium-grain multicomputers are compared and discussed. The paper of Nevanlinna centers on more theoretical aspects of Waveform Relaxation for high performance circuit simulation.

Actual implementations of the algorithms described here are discussed for the iPSC/2 and Symult s2010 multicomputers.



log N/G: Nodes normalized by Computational Grains

The Concurrency Diagram illustrates the trade-offs between the "best" parallelized sequential algorithm and the "best" concurrent algorithm. The former has a higher sequential fraction, but lower overhead compared to the latter. The "best" concurrent algorithm has additional (parallelizable) overhead, but a smaller sequential fraction, allowing it to achieve higher speedups when many nodes are used (large-resource limit, beyond ρ^*).

Waveform Relaxation Methods for Solving Parabolic Partial Differential Equations

Stefan Vandewalle*

Department of Computer Science, Katholieke Universiteit Leuven Celestijnenlaan 200A, B-3030 Heverlee, Belgium

Abstract

The numerical solution of a parabolic partial differential equation is usually calculated by a time stepping method. This precludes the efficient use of vectorization and parallelism, if the problem to be solved on each time level is not very large. In this paper we present an algorithm which overcomes the limitations of the standard marching schemes by solving for the solution on all the time levels simultaneously. The method is applicable to linear and nonlinear problems on arbitrary domains. It can be used to solve initial-boundary value problems as well as time-periodic equations.

1. Introduction

Standard parabolic marching schemes are generally classified as either explicit, implicit or semi-implicit, depending on the discretization of the time derivative. We have compared the parallel characteristics of several classical techniques in [13]. The results can be summarized as follows.

- The explicit methods are highly parallel. Parallel efficiencies close to optimal can easily be obtained. They suffer however from a severe stability constraint, which necessitates the use of very small time steps and makes them less attractive for solving large problems.
- The implicit methods transform the problem into an elliptic partial differential equation that has to be solved on each time level. The multigrid method can be used to solve these equations very rapidly, see [3]. This method uses a hierarchy of fine and coarse grids. The fine grid operations can be performed very efficiently on a multiprocesser. It is much more difficult to parallelize the coarse grid operations since parallel overheads cannot be neglected, see e.g. [11].

• In the semi-implicit methods, the problem of solving one very large system of equations for each time step is reduced to the problem of solving many decoupled tridiagonal systems. Various parallel algorithms are based on substructuring and cyclic reduction. Their arithmetic complexity is approximately twice that of the best sequential algorithm. This limits their parallel efficiency.

Each of the methods can be parallelized efficiently for problems that are large enough. In that case the best sequential algorithm is also the best parallel one. For (relatively) small problems, only the explicit methods retain their parallel efficiency. However, they are limited by the stability constraint and therefore not competitive. The best standard methods, which are of second order implicit type, perform unsatisfactorily. They suffer from a high communication complexity and hardly take advantage of the available parallel computing power.

New algorithms are therefore needed for solving parabolic problems on large scale parallel machines. These algorithms should either improve the numerical quality of the explicit methods or increase the parallel efficiency of the fast implicit methods. The latter can be obtained by calculating the solution on several or all time levels at once. The waveform relaxation technique, to be presented in section 2, belongs to this class. We will discuss its application for solving initialboundary value problems in section 3. In section 4 we will consider the solution of time-periodic parabolic equations. We have implemented the method on an Intel hypercube. Some implementation aspects will be discussed in section 5. In section 6 we will illustrate the method by two examples and compare its performance to that of a parallel implementation of the best standard method.

Research assistant, National Science Foundation (Belgium)

2. The waveform relaxation method

Waveform relaxation (WR), also called dynamic iteration or Picard-Lindelöf iteration [8], is a technique for solving large systems of ordinary differential equations. We will explain the method by its application to the following system.

$$\frac{d}{dt}y_i = f_i(t, y_1, ..., y_N)$$
 (2.1)

with $y_i(t_0) = y_{i0}$, i=1,...,N for $t \in [t_0, t_f]$. The Jacobi variant of the WR algorithm can be formulated as follows.

$$\begin{split} n &:= 0 \\ \text{choose } y_i^{(0)}(t) \quad \text{for } t \in [t_0, t_f] \text{ and } i = 1, ..., N \\ \text{repeat} \\ \text{for each } i: \\ \text{solve } \frac{d}{dt} y_i^{(n+1)} = f_i(t, y_1^{(n)}, ..., y_{i-1}^{(n)}, y_i^{(n+1)}, y_{i+1}^{(n)}, ..., y_N^{(n)}) \\ \text{with } y_i^{(n+1)}(t_0) = y_{i0} \\ n &:= n+1 \\ \text{until convergence.} \end{split}$$

The adaptation of the algorithm to obtain a Gauss-Seidel or SOR type iteration is straightforward. In the iteration step each differential equation is solved as an equation in one unknown. As such the method is very similar to the iterative techniques for solving algebraic systems.

The theoretical foundations of the WR method have been discussed in a number of papers. In [14] convergence is proven for nonlinear systems. The authors concentrate on the systems of ordinary differential equations that arise in the problem of simulating VLSI devices. For these systems the method has shown to be very effective. An analysis for linear systems is given by Miekkala and Nevanlinna in [8]. Further convergence results are given in [5], in which the relation is established between the number of iterations and the accuracy order of a partially converged solution.

3. Initial boundary value problems

3.1 Standard waveform relaxation

We consider the following parabolic equation

$$\frac{\partial u}{\partial t} + L(u) = f_1 \quad (\mathbf{x}, \mathbf{t}) \in \Omega \ \mathbf{x} \left[\mathbf{t}_0, \mathbf{t}_f \right] \tag{3.1.a}$$

$$B(u) = f_2 \qquad (x,t) \in \partial \Omega \ x [t_0,t_f] \qquad (3.1.b)$$

$$u(x,t_0) = u_0 \qquad \mathbf{x} \in \Omega \tag{3.1.c}$$

where $\Omega \subset \mathbb{R}^n$, L is an elliptic, possibly non-linear operator and B is the boundary operator. After spatial discretization and incorporation of the boundary conditions, the parabolic problem is transformed into a system of ordinary differential equations with one equation at each grid point.

$$\frac{dU}{dt} + L(U) = F$$
, $U(t_0) = U_0$. (3.2)

U is the vector of unknown functions defined at the grid points. L is the operator derived from L by discretization and F is the vector of functions determined by f_1 and f_2 .

The standard WR algorithm may be applied to solve (3.2). For instance, in the case of a five-point finite difference discretization of the heat equation, and with use of the Jacobi algorithm, the equation to be solved at each grid point (x_i, y_i) is written as

$$\frac{d}{dt}u_{ij}^{(n+1)} - \frac{1}{(\Delta x)^2}(u_{i+1,j}^{(n)} + u_{i-1,j}^{(n)} - 4u_{ij}^{(n+1)} + u_{i,j+1}^{(n)} + u_{i,j-1}^{(n)}) = f_{ij}.$$

This is a simple first order differential equation which can be solved using any standard, stiff ODE integrator.

Attempts to use WR in the way described above, to solve parabolic problems have not been very successful. This is due to the slow convergence of the method. Indeed, as was shown in [8], the convergence rates for the Jacobi and Gauss-Seidel scheme are of order $1-O(h^2)$, where h is the mesh size parameter. In contrast to the case of a linear system of equations arising from a discretized elliptic equation, overrelaxation in a SOR fashion does not lead to significantly improved convergence characteristics.

3.2 Multigrid Waveform Relaxation

3.2.1 General Idea. The convergence can be accelerated if WR is combined with the multigrid idea [7,10,12]. For a description of the standard multigrid method we refer to [3]. The method differs from the other iterative techniques in that it uses a set of nested grids, with the finest one corresponding to the one on which the solution is desired. Its superior convergence characteristics are based on the interplay of fine grid smoothing, which annihilates high frequency errors, and coarse grid correction, which is applied to reduce the low frequency errors.

The method is extended to time dependent problems in the following way. Each of the

multigrid operations is adapted to operate on the entire functions $u_{ij}(t)$ instead of on single scalar values.

- The smoothing is performed by applying one or more Gauss-Seidel or damped Jacobi waveform relaxations. Smoothing rates for these relaxations have been given in [7].
- ullet The defect of an approximation \overline{U} is defined as

$$D = \frac{d}{dt}\overline{U} + L(\overline{U}) - F. \tag{3.3}$$

The calculation of the derivative in the computation of the defect can be avoided. The application of a standard WR step to an approximation $U^{(n)}$, resulting in an improved approximation $U^{(n+1)}$, corresponds to a calculation of the following type,

$$\frac{d}{dt}U^{(n+1)} + NU^{(n+1)} = MU^{(n)} + F.$$
 (3.4)

where N and M satisfy L = N - M. The defect,

$$D = \frac{d}{dt}U^{(n+1)} + LU^{(n+1)} - F, \qquad (3.5)$$

can then be calculated easily as.

$$= M (U^{(n)} - U^{(n+1)}). (3.6)$$

• The restriction and prolongation are calculated using identical formulae as in the elliptic case. However these formulae now operate on functions instead of on single values. As an example, we formulate the two-dimensional WR full-weighting restriction operator, in stencil notation,

$$u_{IJ}(t) = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} u_{ij}(t), \qquad (3.7)$$

where $u_{II}(t)$ and $u_{ij}(t)$ are corresponding grid point functions on the coarse and the fine grid.

We will first state the equivalent algorithm of the multigrid correction scheme, which is used for solving linear problems. The equivalent of the full approximation scheme for solving nonlinear problems will be given afterwards.

3.2.2 Multigrid correction scheme. Let G_i , $i = 0, 1, \dots, k$ be the hierarchy of grids with G_k the finest grid and G_0 the coarsest grid. Equation (3.2) or equivalently,

$$\frac{dU_k}{dt} + L_k U_k = F_k, \quad U_k(t_0) = U_{k0}$$
 (3.8)

is solved by iteratively applying the following

algorithm to an initial approximation of Uk.

procedure $mgm (k, F_k, U_k)$ if (k = 0): solve $\frac{d}{dt}U_0 + L_0 U_0 = F_0$ exactly else - perform v_1 smoothing operations

- compute the defect: $D_k := \frac{d}{dt} U_k + L_k \; U_k F_k$
- project the defect on G_{k-1} : $F_{k-1} := I_k^{k-1} \ D_k$
- solve on G_{k-1} : $\frac{d}{dt}U_{k-1} + L_{k-1}U_{k-1} = F_{k-1}$ repeat γ_k times $mgm(k-1,F_{k-1},U_{k-1})$, starting with $U_{k-1} := 0$.
- interpolate the correction to G_k and correct U_k : $U_k := U_k I_{k-1}^k \ U_{k-1}$
- perform ν_2 smoothing operations endif

The algorithm is completely defined by specifying the grid sequence G_i , i=0,...,k, the discretized operators L_i , the inter-grid transfer operations I_{i-1}^i and I_i^{i-1} , the nature of the smoothing relaxations, and by assigning a value to the constants ν_1 , ν_2 and γ_i . So-called V- and W-multigrid-cycles are obtained with the values 1 and 2 for γ_i . Another choice leads to the F-cycle. The algorithm can be combined with the idea of nested iteration. The initial approximation to the problem on G_i is then derived from the solution obtained on G_{i-1} . This leads to the waveform equivalent of the full multigrid method.

3.2.3 Full approximation scheme. The algorithm was extended to nonlinear parabolic problems in [10]. The nonlinear algorithm is easily derived from the well-known multigrid full approximation scheme and is presented at the end of this section.

The derivative calculation in the determination of the coarse grid problem right hand side can be avoided. Indeed, when the two restriction operators I_k^{k-1} and \overline{I}_k^{k-1} are equal, the two derivatives cancel. The right hand side of the problem on the coarse grid may then be calculated by the following formula.

$$F_{k-1}:=L_{k-1}(\overline{U}_{k-1})-I_k^{k-1}(L_k(U_k)-F_k). \eqno(3.9)$$

procedure fas (k, F_k, U_k) if (k = 0): solve $\frac{d}{dt}U_0 + L_0(U_0) = F_0$ exactly

else

- perform ν_1 smoothing operations

- project U_k onto G_{k-1} : $\overline{U}_{k-1} := \overline{I}_k^{k-1}U_k$ - calculate the coarse problem right hand side: $F_{k-1} := \frac{d\overline{U}_{k-1}}{dt} + L_{k-1}(\overline{U}_{k-1}) - I_k^{k-1}(\frac{dU_k}{dt} + L_k(U_k) - F_k)$ - solve on G_{k-1} : $\frac{d}{dt}U_{k-1} + L_{k-1}(U_{k-1}) = F_{k-1}$ repeat γ_k times fas $(k-1, F_{k-1}, U_{k-1})$,

starting with $U_{k-1} := \overline{U}_{k-1}$.

- interpolate the correction to G_k and correct U_k : $U_k := U_k + I_{k-1}^k (U_{k-1} - \overline{U}_{k-1})$ - perform ν_2 smoothing operations
endif

4. Time-periodic parabolic problems

4.1 The standard algorithms

In this section we will consider the parabolic problem (3.1a-c) where the initial condition (3.1.c) is replaced by the periodicity condition

$$u(x,t_0) = u(x,t_t).$$
 (4.1)

This problem is of considerable importance in various areas of practical interest, such as wing flutter, ferro-conductor eddy currents, chemical reactor theory, pulsating stars, and fluid dynamics. Various algorithms have been proposed to compute the stable periodic solutions. One approach is a timeintegration of the studied system, starting from an arbitrary initial condition, until a stable periodic orbit is reached. This brute force method may however be prohibitively expensive in the case of slowly decaying transients. A second approach consists of using difference methods where a large system of nonlinear algebraic equations is obtained after discretization. This system may be solved with direct or iterative sparse solvers, [9]. A third and commonly used approach is based on the shooting method [4]. Finally, a very fast algorithm was presented by Wolfgang Hackbush in [2], in which the periodic problem is reformulated as an integral equation and solved by the multigrid method of the second kind. We will briefly review this algorithm. In section 6, it will be used to compare a new, WR based algorithm with. We will restrict our attention to the linear case as the nonlinear algorithm is very similar.

4.2 Multigrid method of the second kind

The solution of the linear initial-boundary value problem (3.1.a-c), restricted to t_f , can be written as the outcome of an affine mapping applied to the initial condition u_0 ,

$$u_f = K u_0 + f. (4.2)$$

K is a linear integral operator, such that $K u_0$ equals $u(x,t_f)$, the solution to (3.1.a-c) with homogeneous right hand sides (f_1 =0 and f_2 =0), while f(x) equals $u(x,t_f)$, the solution to (3.1.a-c) with zero initial condition (u_0 =0). With this notation, the periodicity condition (4.1) becomes

$$y = K y + f, \tag{4.3}$$

where y(x) is a function on Ω . The determination of a function y satisfying (4.3) is equivalent to the problem of finding a function u that satisfies (3.1.a-b) and (4.1). Indeed, if y fulfills (4.3), then the solution u of the initial boundary value problem (3.1.a-c) with $u_0 = y$, is the solution of the time-periodic problem.

(4.3) is a Fredholm integral equation of the second kind and may be solved by the very fast multigrid method of the second kind. We refer to [3] for an in depth analysis of this technique and for a discussion of various applications. In a similar way as in the multigrid method for elliptic equations, (4.3) is discretized on a set of grids, G_i , i=0,...,k, resulting in a set of discrete equations,

$$Y_i = K_i Y_i + F_i, \quad \text{on } G_i. \tag{4.4}$$

The problem on the fine grid is solved by iteratively applying the following algorithm to an initial approximation of Y_k .

procedure mgm_2nd (k,F_k,Y_k) if (k=0): solve $Y_0 = K_0 Y_0 + F_0$ exactly

else

- smoothing: $Y_k := K_k Y_k + F_k$ - compute the defect: $D_k := Y_k - K_k Y_k - F_k$ - project the defect on G_{k-1} : $F_{k-1} := I_k^{k-1} D_k$ - solve on G_{k-1} : $Y_{k-1} = K_{k-1} Y_{k-1} + F_{k-1}$ repeat 2 times mgm_2nd $(k-1,F_{k-1},Y_{k-1})$,

starting with $Y_{k-1} := 0$.

- interpolate the correction to G_k and correct U_k : $Y_k := Y_k - I_{k-1}^k Y_{k-1}$ endif

No explicit representation of the discretized integral operator K_i is required. Indeed, application of K_i to a function Y_i is equivalent to calculating the solution of one discrete initial-boundary value problem defined on G_i . K_i Y_i may thus be computed by using standard parabolic solvers, such as a time-stepping method, or by using the waveform relaxation algorithm of section 3.

In [3] the convergence rate of the algorithm is shown to be of the order $O((\Delta x_k)^2)$, where Δx_k is the fine grid mesh size. As such, one iteration step is usually sufficient to solve (4.3) to discretization accuracy. (To obtain this result, some mild restrictions on the size of time increment, Δt_i , i=0,...,k, have to be taken into account, in order to guarantee a sufficient smoothing behavior of the time discretization formula.) It can easily be shown that the arithmetic complexity of one iteration of the algorithm is of the same order as the complexity of solving an initial-boundary value problem on the fine grid.

4.3 A waveform relaxation algorithm

Spatial discretization of (3.1.a-b) and (4.1) leads to the following system of ordinary differential equations,

$$\frac{dU}{dt} + L(U) = F$$
, $U(t_0) = U(t_f)$. (4.5)

This system may be solved with a waveform relaxation algorithm that is only slightly different from the algorithm discussed in section 2. Instead of repeatedly solving an ordinary differential equation of initial value type at each grid point, one repeatedly solves the following periodic differential equation

$$\frac{du_{ij}}{dt} + (L(U))_{ij} = f_{ij}, \ u_{ij}(t_0) = u_{ij}(t_f). \tag{4.6}$$

This problem may be solved e.g. by a discretization method, resulting in a sparse matrix equation. Application of a implicit one-step discretization method leads to an easily solvable, almost bidiagonal matrix equation.

The modified waveform relaxation can be used as such, or can be integrated as a smoother into any of the multigrid schemes of section 3. Numerical evidence shows that the latter leads to a rapidly converging iteration, with typical multigrid convergence rates.

5. Implementation aspects

5.1 Parallelization

We have implemented the WR algorithms on an Intel iPSC/2-VX hypercube. For a description of this multiprocessor, its hardware characteristics and various performance benchmarks, we refer to [1]. The implementation is discussed in great detail in our studies [11.12.13]. In this paper we will only go over some of the main issues.

A classical data decomposition is used to evenly distribute the computational workload. The processors are arranged in a rectangular array and are mapped onto the domain of the partial differential equation. Each processor is responsible for doing all computations on the grid points in its part of the physical domain. During the computation, communication with neighboring processors is needed to update local boundary values. Various other communications strategies may further be used to improve the parallel performance. We want to mention in particular the use of an agglomeration strategy to reduce the communication complexity of the coarse grid operations, [11].

In the WR method each grid point is associated with an unknown function, uit(t). In our implementation, such a function is represented as a vector of function values evaluated at equidistant time levels. We denote the vector length by n₁ (number of time intervals). The arithmetic complexity increases linearly with the value of n_t. In the same way, the total length of the messages exchanged during the computation is proportional to the vector length. The number of message exchanges. however, and the sequential overhead due to program control are independent of nt. From the high message startup time on most parallel machines (and in particular on the iPSC/2), it is clear that the communication time to calculation time ratio will decrease with increasing function length and that the parallel efficiency will improve.

In figure 3.2, we present typical speedup values (S_p) measured on a 16 processor machine. Two curves are drawn, one for a waveform multigrid cycle, with $n_t = 50$, and one for a standard multigrid cycle, as it is used for solving elliptic problems. (The particular problem that is solved, is discussed in [12].) The substantial performance difference is due to the very different calculation to communication ratios.

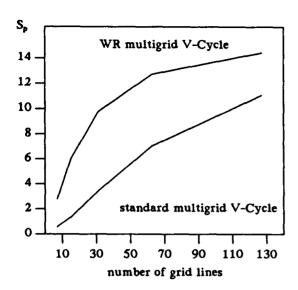


Figure 5.1. Typical speedups for waveform multigrid and standard multigrid cycles

5.2 Vectorization

The use of a vector processor in each computer node may result in a substantial reduction of computing time. Indeed, most of the waveform multigrid operations can be expressed as simple arithmetic operations on functions, i.e. on vectors, see e.g. the restriction operator (3.7). In contrast to the standard approach we do not vectorize in the spatial direction but we vectorize in the time direction. The vector speedup of the arithmetic part of the computation will mainly depend on the value of the vector length parameter nt. It will be virtually independent of the size of the spatial grid, the number of multigrid levels, the multigrid cycle used and the number of processors. This is in sharp contrast with standard multigrid vectorization results, see e.g. [6]. Standard vectorization does not lead to a performance improvement unless the number of grid points per processor is very large. Its application is therefore of very limited use on large scale parallel processors.

As a second advantage of our approach we may mention the ease of implementation. As the vector operations at a each grid point involve the vectors at neighboring grid points only, no complex grid restructuring (as in the standard approach) is needed.

The only operation which is not perfectly vectorizable is the core of the ODE integrator, which is used in the smoothing step, and which is inherently sequential. It will therefore reduce the

possible gain through vectorization. It can be shown that the non-vectorizable part of the calculation makes out at most 10% of the total computation. This leads to a possible vector speedup of 10, or more.

6. Numerical examples

6.1 An initial-boundary value problem

We consider the solution of the following initial-boundary value problem,

$$\frac{\partial \mathbf{u}}{\partial \mathbf{t}} = \frac{\partial^2 \mathbf{u}}{\partial \mathbf{x}^2} + \mathbf{x}\mathbf{y} \frac{\partial^2 \mathbf{u}}{\partial \mathbf{x} \partial \mathbf{y}} + \frac{\partial^2 \mathbf{u}}{\partial \mathbf{y}^2} + \mathbf{f}$$
 (6.1)

defined on $\Omega = [0,1]x[0,1]$ for $t \in [0,0.5]$, with Dirichlet conditions on the northern, eastern and southern boundary and a Neumann condition on the boundary to the west. The right hand side function f is chosen in such a way that the solution is equal to

$$u(t,x,y) = \sin(5x+y+10t) e^{-4t}$$
.

For this problem we will compare the performance of the WR method with a parallel implementation of the "best" sequential method, the Crank-Nicolson method. Our implementations of both methods are highly optimized and are of similar complexity. In both cases multigrid is used with a four-color nine-point Gauss-Seidel smoother, standard coarsening to a 3 by 3 coarse grid, full weighting restriction, bilinear interpolation and a coarse grid solver that performs 2 Gauss-Seidel iterations. A constant time step, Δt , is chosen for the Crank-Nicolson method, similar to the time step used to represent the functions in the WR method. In this example, Δt was set to 0.01, independent of the spatial discretization. This leads to a vector length of 50. In the WR method we use the trapezoidal rule to solve the differential equations.

Some results, obtained on a 16 processor machine, are depicted in figure 6.1. The graphs show the accuracy of the solution (largest error at the grid-points) versus execution time. The figures show smooth curves for the WR method. The error of the initial waveform approximation gradually decreases as more and more multigrid cycles are applied. The Crank-Nicolson results show up as discrete points. The Crank-Nicolson solution process is advanced time step per time step in a total of t seconds. The accuracy of the result is represented by a "+" -sign at position (t,error) in the figure.

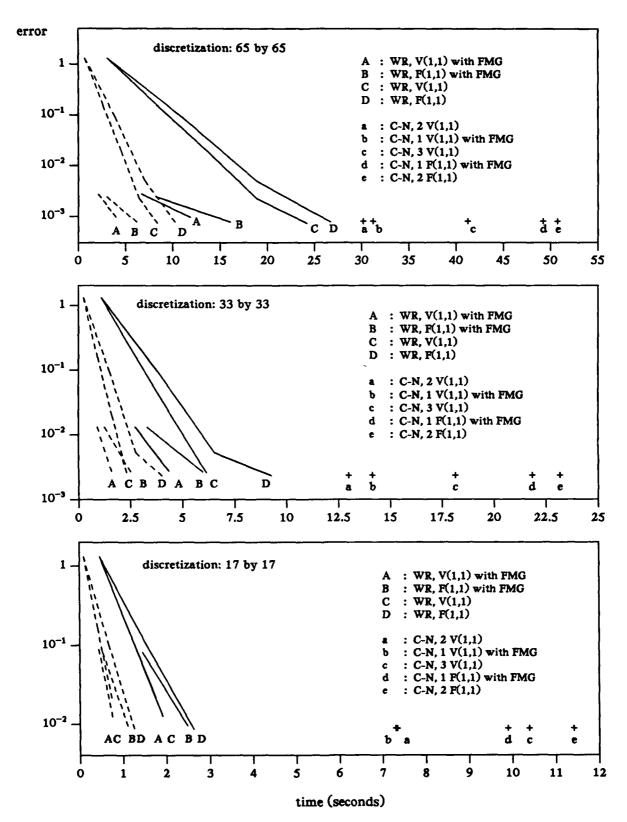


Figure 6.1. Comparison of Crank-Nicolson and WR Multigrid execution times

Table 6.1: Execution time, in seconds, of the full multigrid solver on 1 and 16 processors							
	17	by 17 problem		33 by 33 prob		lem	
method	1 proc.	16 proc.	Sp	1 proc.	16 proc.	Sp	
Waveform Relaxation	9.53	1.91	5.0	36.00	4.30	8.4	
Crank-Nicolson	15.89	7.34	2.2	58.72	14.23	4.1	

	Table 6.2: Execution time, in seconds, of the WR FMG V(1,1) solver (on 16 processors)								
	17 by 17 problem 33 by 33 p			by 33 proble	m	65 by 65 problem		m	
nt	scalar	vector	Sp	scalar	vector	Sp	scalar	vector	Sp
100	3.54	1.12	3.16	8.20	2.42	3.39	(na)	(na)	(na)
50	1.91	0.76	2.51	4.34	1.59	2.73	11.88	4.02	2.96
25	1.10	0.59	1.86	2.43	1.20	2.03	6.53	2.91	2.24
10	0.62	0.49	1.27	1.29	0.97	1.33	3.21	2.25	1.43

Depending on the cycle type used, different execution times are needed. As such several results are presented for each technique. They are annotated in figure 6.1 in the following way: with "WR F(1,1) with FMG" we mean "waveform relaxation using F-cycles with 1 pre- and 1 post-smoothing step and the full multigrid technique with 1 cycle at each grid level"; with "C-N, 2 F(1,1)" we mean "Crank-Nicolson method with 2 F(1,1) cycles per time step". Two sets of curves are given for the WR method. The dashed lines represent the results obtained with vectorization, while the solid lines represent the results obtained in scalar execution mode.

On 16 processors, WR turns out to be faster than the Crank-Nicolson method by a factor of 8 (for the 65 by 65 problem) up to a factor of 10 (for the 17 by 17 problem). This is due to the smaller arithmetic complexity of the waveform method, its superior parallel characteristics and the use of vectorization.

In table 6.1 we have tabulated the execution time of the full multigrid solver with one V(1,1) cycle on each grid level, on 1 and on 16 processors. We have also added the parallel speedup. Sp. Waveform relaxation outperforms the standard method by a factor of approximately 1.7, on a single processor. This is due to the smaller arithmetic complexities of the smoothing and the defect calculation steps (which account for a factor of approximately 1.5), a reduced initialization cost (some intermediate results may be retained when setting up system (3.2)) and the lower computational

overheads associated with program control (loop overhead, indexing overhead, procedure call overhead, etc.). An additional factor of 2, and higher, results from the better parallel characteristics of WR method. This is easily seen from the speedup figures.

The remaining performance difference is due to vectorization. In table 6.2 we give the execution times of the WR full multigrid solver. The dependence of the vector speedup on the vector length is obvious. It should be noted that for problems of this size on a 16 processor machine, standard vectorization in the Crank-Nicolson method would not lead to any speedup [6].

6.2 A time-periodic problem

We consider the parabolic partial differential equation,

$$\frac{\partial \mathbf{u}}{\partial \mathbf{t}} = \frac{\partial^2 \mathbf{u}}{\partial \mathbf{x}^2} + \frac{\partial^2 \mathbf{u}}{\partial \mathbf{v}^2} + \mathbf{f} \tag{6.2}$$

with the following time periodicity condition, u(0,x,y) = u(1,x,y), defined on the unit square with four Dirichlet boundary conditions. The function f is chosen such that the solution of the PDE equals

$$u(t,x,y) = (x-x^2)^2(y-y^2)^2\sin(2\pi t)$$

In figure (6.2) we represent the timing results obtained on a 16 processor hypercube. (No vectorization was used for this example.)

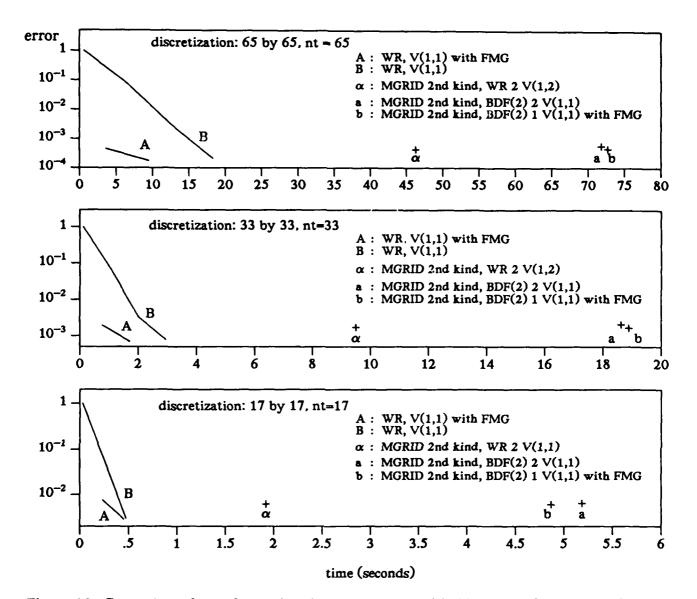


Figure 6.2. Comparison of waveform relaxation method and multigrid method of the second kind

Three methods are compared. The first method is a parallel implementation of the multigrid method of the second kind, as proposed by Hackbush. The second order backward differentiation method (BDF(2)) is used for time integration. It has excellent smoothing properties and is of high accuracy. The mesh size on each grid G_i , Δx_i , is determined by standard coarsening from a fine grid, G_k , with 65, 33, or 17 grid lines in x- and y-direction. The time increment, Δt_i , is chosen equal to the mesh size. The linear systems obtained by the BDF(2) scheme on each time-level are solved by using the standard multigrid method, with the 2 V(1,1) cycles or full multigrid. Thanks to its $O((\Delta x)^2)$ convergence rate, one iteration of the

method is sufficient to solve the problem to discretization accuracy.

Various programming techniques are applied to optimize the parallel performance of the implementation. In particular, an agglomeration technique is used to reduce the parallel overhead of the coarse grid operations, [11].

A related method is obtained if the multigrid WR algorithm is used as the smoother inside the multigrid method of the second kind. The resulting algorithm is between 2 and 3 times as fast the method with time stepping, as can been seen in fig. 6.2. This was to be expected, from the results of section 6.1.

The periodic multigrid WR method shows to be faster than the best standard algorithm, by a factor of 7 to 10. This is in part due to its lower arithmetic complexity. If the complexity of solving the initial-value problem (3.1.a-c) on the fine grid G_k is denoted by Wk, it may be shown that the cost of the multigrid algorithm of the second kind is approximately 2.5 Wk, whereas the execution of a full multigrid WR step is only 1 Wk. The remaining performance difference results from the better parallel characteristics of the WR method. As was noted in the introduction, it is difficult to efficiently parallelize coarse grid operations. The multigrid method of the second kind visits the coarse grid very frequently, because of its "double multigrid" nature. It is basically a multigrid Wcycle, where, in each smoothing step, a large number of elliptic problems are solved by standard multigrid. Consequently, the algorithm is not well suited for parallel implementation.

We should also note that vectorization will lead to an additional speedup, in the case of the WR algorithm only. The performance difference on the 16 processor machine will then be in the range of 25 to 50, depending on the problem size.

7. Concluding remarks

The transformation of the parabolic problem into the sequential process of solving small problems on successive time levels, seriously degrades parallel efficiency of the standard marching schemes. While they can be used efficiently for problems with a very large number of grid points per processor, they perform totally unsatisfactorily for small problems and large numbers of processors.

We have presented several methods based on waveform relaxation. They show multigrid convergence speeds and can be efficiently implemented on parallel machines. As an added advantage they can be straightforwardly vectorized even if the number of grid points per processor is very small. As such they are perfectly fit for implementation on massively parallel machines.

References

- [1] Bomans, L., and Roose, D., Benchmarking the iPSC/2 hypercube multiprocessor, Concurrency: Practice and Experience, 1 (1) 1989, pp. 3-18.
- [2] Hackbush, W., Fast numerical solution of timeperiodic parabolic problems by a multigrid method, SIAM J. Sci. Stat. Comput., Vol. 2., No. 2, 1981,

pp. 198-206.

[3] Hackbush, W., Multi-grid methods and Applications, Springer Series in Comp. Math. 4, Springer-Verlag, Berlin, 1985.

[4] Holodniok, M. and Kubicek, M., DERPER - an algorithm for continuation of periodic solutions in ordinary differential equations J. Comput. Phys. 55, 1984, pp. 254-267.

[5] Juang, F., Accuracy increase in waveform relaxation, Report No. UIUCDCS-R-88-1466, department of computer science, university of Illinois at Urbana-Champaign, Urbana, Illinois, Oct. 1988.

[6] Lemke, M., Experiments with a vectorized multigrid Poisson solver on the CDC 205, Cray X-MP and Fujitsu VP200, Arbeitspapiere der GMD 179, GMD, Sankt Augustin, Nov. 1985.

[7] Lubich, Ch. and Ostermann, A., Multi-Grid Dynamic Iteration for Parabolic Equations, BIT, 27 (1987), pp. 216-234.

[8] Miekkala, U. and Nevanlinna, O., Convergence of Dynamic Iteration Methods for Initial Value Problems, SIAM J. Sci. Stat. Comput., Vol. 8, No. 4, 1987, pp. 459-482.

[9] Tee, G.J., An Application of p-Cyclic Matrices, for Solving Periodic Parabolic Equations, Num. Math. 6, 1964, pp. 142-159.

[10] Vandewalle, S. and Roose, D., The Parallel Waveform Relaxation Multigrid Method, in: Rodrigue, G., (ed.), Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1989, pp. 152-156.

[11] Vandewalle, S. and Piessens, R., A Comparison of Parallel Multigrid Strategies, in: Andre, F. and Verjus, J.P., (eds.), Hypercube and distributed computers, North-Holland, Amsterdam, 1989, pp. 65-79.

[12] Vandewalle, S. and Piessens, R., A Comparison of the Crank-Nicolson and Waveform Relaxation Multigrid Methods on the Intel Hypercube, in: Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods, J. Mandel, S. McCormick, J. Dendy, C. Farhat, G. Lonsdale, S. Parter, J. Ruge and K. Stuben (eds), SIAM, Philadelphia, 1990, pp. 417-434.

[13] Vandewalle, S., Van Driessche, R. and Piessens, R., The Parallel Implementation of Standard Parabolic Marching Schemes, report TW125, Katholieke Universiteit Leuven, Dec. 1989.

[14] White, J., Odeh, F., Sangiovanni-Vincentelli, A.S. and Ruehli, A., Waveform Relaxation: Theory and Practice, Memorandum No. UCB/ERL M85/65, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1985.

A Parallel Implementation of ESACAP

Stig Skelboe
Department of Computer Science
University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen

Abstract

The DC analysis and transient analysis parts of the general electrical circuit analysis program ESACAP are parallelized to run on the Intel iPSC. Most of the program runs unchanged on the cube manager. Only the solution of the systems of nonlinear algebraic equations is parallelized to run on the hypercube parallel computer. The nonlinear equations arise either from the DC problem or from the discretization of the differential equations by backward differentiation formulas.

Circuit equations are allocated to processors to balance the load of function evaluation and LU-factorization algorithm used by the Newton iteration algorithm. Using p processors, a speed-up of approximately p/2 can be obtained. The performance of the parallel program is demonstrated by simulating a 4×4 bit digital multiplier circuit leading to 180 circuit equations.

1 Techniques for Parallel Circuit Analysis

The time domain analysis of analogue circuits or digital circuits at the circuit level involves the numerical solution of systems of nonlinear ordinary differential equations. The differential equations are usually stiff which implies that implicit numerical integration formulas must be used. Besides, models of electrical circuits often lead to coupled systems of differential and algebraic equations, and consequently each time step involves the solution of a system of nonlinear algebraic equations including the discretization of the differential equations.

Let a circuit be described by the following implicitly given system of differential algebraic

equations,

$$f(t, y, y') = 0 \tag{1}$$

where $f: R \times R^N \times R^N \to R^N$. When discretized by the simple backward Euler formula, the following nonlinear algebraic system is obtained,

$$f(t_n, y_n, (y_n - y_{n-1})/h) = 0$$
 (2)

where h is the stepsize in time, $h = t_n - t_{n-1}$ and $y_n \approx y(t_n)$.

The solution of (2) is obtained by some iterative method, usually of Newton type,

$$y_n^{(m+1)} = y_n^{(m)} - F_n'(y_n^{(m)})^{-1} f_n(y_n^{(m)})$$
 (3)

where $f_n(y) = f(t_n, y, (y - y_{n-1})/h)$ and $F'_n(y) = \partial f_n(y)/\partial y$.

The main computational task involved in the Newton iteration is the evaluation of the nonlinear vector function f and the nonlinear matrix function F'. The matrix F' is generally sparse, and this is exploited in the solution of the linear equations indicated by the matrix inverse. The computational complexity of the linear equation solution is proportional to N^k where k is in the interval from 2 to 3 and N is the dimension of the problem (1). The solution of the linear equations therefore becomes relatively larger compared with the evaluation of f and F' when N increases.

The obvious approach to a parallel version called direct parallelization is to compute f and F' in parallel and solve the linear equations in parallel. This simply amounts to a parallel version of the Newton iteration (3). The vector and matrix functions parallelize well, but the general sparse system of linear equations is hard to solve efficiently on a parallel computer.

An alternative approach to a parallel numerical solution of (1) is the so called waveform relaxation [1],[2]. In this approach,

the system (1) is decomposed into a number of loosely coupled subsystems of differential equations, and each subsystem is solved independently over the same time window. The process is repeated a number of times, and after each iteration, information is exchanged between the subsystems.

Waveform relaxation was not intended primarily for parallel computers, but it was observed in [2] that the method maps well on medium grain distributed memory computers, like the Intel iPSC Hypercube. Waveform relaxation is very efficient for the simulation of digital MOS circuits, and it has been attempted for a broader class of problems. However, the same gain in efficiency is not always obtained and there may be problems with convergence.

In the parallel version of waveform relaxation, one or several subsystems are allocated to each processor. For certain problems, the subsystems become so large that it is necessary to apply several processors to solve one subsystem. Otherwise, the result would be very poor load balance. In this case, the direct parallelization approach can be applied to a subsystem.

This paper exploits the possibilities of direct parallelization for the following reasons. If an efficient approach can be devised, it can be used in the parallelization of numerous existing simulation programs like ESACAP. The balance between computations which parallelize well and computations which do not parallelize well may turn out to be favourable, leading to a good overall speed-up. Last, direct parallelization of large blocks in a waveform relaxation program can extend the class of problems for which this very powerful approach is useful.

2 ESACAP

ESACAP [3] is a general purpose circuit analysis program primarily developed for DC, transient and periodic steady-state analysis of nonlinear electrical circuits. For linear circuits, it also offers frequency response and zero/pole computation. The circuits are described in a flexible input language which permits the specification of general nonlinear relationships. The Jacobian F' is computed analytically by ESACAP on the basis of the input

specification.

The circuit description is transformed into modified nodal equations [4] which are reordered to fill diagonal zeros. Then the Jacobian F' is reordered to reduce fill-ins during LU-factorization. The DC problem is solved by a hybrid method [5], and the differential equations are integrated numerically by backward differentiation formulas automatically selected from orders 1 to 6.

Figure 1 shows an example of input for ESACAP, a full adder realized with transmission gates [6]. The circuit description is hierarchical with the transistor model at the lowest level, then the gates and finally the adder. The transistors can be modelled according to the needs, but usually a suitable model is found in a library.

The problem used for benchmarks throughout this paper is a corrected version of the multiplier found on p. 345 in [6]. It is composed of full adders as specified in Figure 1, half adders trivially derived from the full adder and simple C-MOS gates.

3 Parallel ESACAP - Outline

In this project, only the numerical solution of ordinary differential equations (transient analysis) has been chosen for parallelization. The backward differentiation formulas used in ESACAP do not permit parallelization "across the method" but only parallelization "across the system" [7]. This means that the system of equations (1) is partitioned into groups of equations where each group is assigned to a different processor.

The original version of ESACAP runs on the iPSC Cube Manager. In order to simplify the modifications, it was decided to confine the parallelization to the subroutine implementing Newton iteration (3). The original Newton iteration performed for each time step of the numerical integration is substituted by a subroutine providing data for the 32 node iPSC which runs a parallel Newton iteration loop.

The iteration scheme (3) is implemented in three different versions. The first version is a true Newton iteration as specified by (3). The second version is a pseudo Newton iteration where $F'_n(y_n^{(0)})$ is used through all iterations

```
$$DES
                                           MPTYP*V(GATE, SOURCE)-UTR,
# AUXILIARY FUNCTIONS
                                           MPTYP+V(DRAIN, SOURCE), BETA, LAMBD);
$FUNCTION: RAMP(X.A.B.G.E):
                                           # CAPACITANCES
  ARG=EXP(A*(X-G));
                                           CGADR(GATE, DRAIN) = CGD3(
  RAMP=B*((ARG-1)/(ARG+1))+E;
                                           MPTYP*V(GATE, SOURCE),
                                           MPTYP*V(DRAIM, SOURCE), UTR, COX, CGDO);
$FUNCTION: AF1(UGS.UT.APHI):
                                           CGASO(GATE.SOURCE)=
  AF1=RAMP(UGS,8/APHI,1/3,
                                           CGS3(NPTYP+V(GATE, SOURCE),
  UT-APHI/4,1/3);
                                           NPTYP+V(DRAIN.SOURCE).
                                           UTR, COX, APHI, CGSO)+
$FUNCTION: AF2G1(UGS.UT):
                                           CGB3(NPTYP*(V(GATE.SOURCE)~BS).
  AF2G1=RAMP(UGS, 10, 1/4, UT, 1/4);
                                           UTR, COX, APHI, CGBO);
                                           END:
                                           # C-MOS INVERTER
$FUNCTION:
  CGS3(UGS.UDS.UT.COX.APHI.CGSO):
                                           $MODEL: INV(IN.OUT, REF, UDD):
                                           X1(OUT, IN, REF)=MOS2U3(UT=0.9, BETA=40U,
  CGS3=CGSO+COX+IFGT(UDS,O,
                                           GAMMA=0.3, LAMBD=0.05, APHI=0.6,
  AF1(UGS,UT,APHI),AF2G1(UGS,UT));
                                           CGD0=9F,CGS0=9F,CGB0=9F,COX=16F);
# IFGT:
                                           X2(OUT.IN,UDD)=MOS2U3(NPTYP=-1,UT=-0.9,
# IF UDS>0 THEN AF1(.. ELSE AF2G1(..
                                           BETA=40U, GAMMA=.4, LAMBD=.05, APHI=-0.6,
$FUNCTION: CGD3(UGS,UDS,UT,COX,CGDO):
                                           CGDO=9F,CGSO=9F,CGBO=9F,COX=16F);
  CGD3=CGD0+COX*IFLT(UDS.O.
                                           # C-MOS TRANSMISSION GATE
  AF2G1(UGS,UT),0);
# IFLT:
                                           $MODEL: TGATE(IN,OUT,T,NT);
# IF UDS<0 THEN AF2G1(.. ELSE 0
                                           X1(IN, T, OUT) = MOS2U3(UT=0.9, BETA=40U,
                                           GAMMA=0.3, LAMBD=0.05, APHI=0.6.
$FUNCTION: CGB3(UGB, UT, COX, APHI, CGBO);
                                           CGDO=9F,CGSO=9F,CGBO=9F,COX=16F);
  U1=UT-APHI/2;
                                           X2(IN, NT, OUT) = MOS2U3(NPTYP=-1, UT=-0.9,
  CGB3=CGBO+
                                           BETA=40U, GAMMA=.4.LAMBD=.05.APHI=-0.6.
  COX*RAMP(UGB,-1,0.425,U1,0.425);
                                           CGDO=9F,CGSO=9F,CGBO=9F,COX=16F);
END:
                                           # C-MOS FULL ADDER
$FUNCTION:
                                           $MODEL: ADDER(A,B,C,SUM,CARRY,UDD);
  IDS2U3(GSEFF,DS,BETA,LAMBD);
                                           X1(A, NA, NREF, UDD) = INV;
  IDS2U3=IFLT(GSEFF,0,0,IFLT(DS,GSEFF,
                                           X2(B,APB,NA,A)=INV;
  2*BETA*(GSEFF-DS/2)*DS.
                                           X3(B, NAPB, A, NA)=INV;
  BETA*GSEFF*GSEFF)*(1+LAMBD*DS));
                                           X4(B, APB, NA, A)=TGATE;
                                           X5(B, NAPB, A, NA)=TGATE;
# MOS TRANSISTOR MODEL
                                           X6(C.NC.NREF.UDD)=INV:
# FIRST LEVEL SPICE MODEL
                                           X7(NC, NSUM, NAPB, APB) = TGATE:
$MODEL: MOS2U3(DRAIN, GATE, SOURCE):
                                           X8(C, NSUM, APB, NAPB) = TGATE;
NPTYP, UT, BETA, GAMMA, LAMBD, APHI, BS,
                                           X9(NSUM, SUM, NREF, UDD) = INV;
CGDO, CGSO, CGBO, CDX;
                                           X10(NB, NCARRY, NAPB, APB)=TGATE;
# DEFAULT PARAMETERS
                                           X11(NC, NCARRY, APB, NAPB)=TGATE;
DEF(NPTYP=1,UT=1.0,BETA=4.8U,
                                           X12(MCARRY, CARRY, NREF, UDD)=INV;
GAMMA=0.205, LAMBD=0.03, APHI=0.536,
                                           X13(B, WB, WREF, UDD) = INV;
BS=0,CGD0=1F,CGS0=1F,CGB0=1F,COX=1F);
                                           END;
# THRESHOLD VOLTAGE
                                           $$STOP
UTR=UT+NPTYP+
                                           Figure 1: ESACAP description of a transmis-
GAMMA*(SQRT(NPTYP*APHI-NPTYP*BS)-
                                           sion gate adder.
SQRT(NPTYP*APHI));
# DRAIN-SOURCE CURRENT
```

JDS(DRAIN, SOURCE) = NPTYP + IDS2U3(

for y_n . The last version tries to use $F'_n(y_n^{(0)})$ for the computation of $y_n, y_{n+1}, ...$ as far as possible.

If the Newton type iterations fail to converge, the program falls back on a parallel version of the hybrid method [5] primarily designed for DC analysis. This means that parallel DC analysis comes "free".

A number of circuit equations (components of the vector function f) and the corresponding rows of the Jacobian F' are allocated to each processor. The factorization of the linear equations is based on the same processor allocation. The pivot rows are broadcast to all processors one by one, and the elimination takes place in parallel.

The parallelization of ESACAP served three purposes. First, to gain experience with the porting of large sequential programs to a parallel computer. Only a small part of ESACAP had to be modified and other parts rearranged, and this is believed to be a typical situation. The parallelization also involved the splitting of data structures, and in the case of ESACAP this task was nontrivial. Second, to measure the overall speed-up of a parallel nonlinear integration routine for stiff systems of ordinary differential equations. It is very difficult to solve sparse linear equations efficiently in parallel, but the overall speed-up is still acceptable when this part is non-dominant.

Finally, the purpose was to get a test vehicle for testing various parallel sparse linear equation solvers. The powerful input language of ESACAP permits the specification of models of problems from a variety of areas to generate test problems for the equation solver.

4 Functions and Jacobian

The functions specified in the input language of ESACAP are converted into an internal form based on reverse Polish notation. The vector function f_n of (3) is computed from this internal representation, and also the derivatives required for F'_n are computed analytically from the same representation.

The derivatives are computed for the actual $y_n^{(m)}$ vector; no symbolic representation of the derivatives exists. This approach leads to less efficient evaluation of the nonlinear models than the traditional approach where the nonlinear functions and derivatives are imple-

mented as Fortran subroutines. However, it gives the user the ultimate freedom in specifying nonlinear relations, and besides it improves the potential of speed-up from parallelization, because it shifts the computational complexity from the solution of linear equations towards the computation of nonlinear functions.

The fundamental building blocks in the input language of ESACAP are two terminal elements, and the circuit modelled by these elements is represented by modified node equations [4]. This means that each two terminal element in general appears in two equations, or if one terminal is grounded in only one equation. If two node equations including the same nonlinear two terminal element are allocated to two different processors, this results in a duplicate computation of the corresponding function. The minimization of duplicate computation is one of the objectives of processor allocation.

A serious problem of duplicate computation of nonlinear functions and poor load balance may be caused by the modelling of the power supply of a digital circuit. Approximately half of the transistors may be connected to the voltage source modelling the power supply leading to a node equation containing half of the nonlinear functions of the total circuit.

The problem is not handled automatically in the present parallel version of ESACAP. However, it is easily solved manually by duplicating the power supply voltage source sufficiently many times to reduce the the maximum number of functions of the node equations of the voltage sources. The penalty is a larger number of node equations which is a low penalty in this connection.

5 Sparse Matrix Solver

The parallel sparse matrix solver is based on the original sparse matrix solver of ESACAP. Electrical circuits will in general lead to nonsymmetric matrices, but since they are close to being structurally symmetric, they are treated as such by ESACAP. This leads to simpler and more efficient processing of the sparse matrices. The reordering to reduce fill-ins is based on the third scheme of Tinney and Walker [8]. It is a symmetric row and column reordering which preserves diagonal elements in the diagonal.

The reordered matrix is distributed to the processors according to a row interleaved scheme. With p processors, rows k, k + p, k + 2p, ... are allocated to processor k for k = 1, 2, ..., p. This is a standard scheme which assures as uniform load distribution as possible during LU-factorization.

The basic LU-factorization of an $N \times N$ matrix A can be outlined as follows,

```
for i:=1 to N do
  for j:=i+1 to N do
  begin
    eliminate column i
    in row j using row i;
    save pivot element in A[j,i]
end
```

Algorithm 1

The algorithm of processor k executing a parallel LU-factorization based on row interleaved processor allocation has the following outline,

```
for i:=1 to N do
  begin
    {if row i is on processor k ..}
    if i in [k,k+p,k+2p,...] then
      broadcast row i
    else receive row i;
    {row j is the first row on
    processor k where j>i}
    j:=((i+p-k) \text{ div } p)*p+k;
    while j<=N do
      begin
        eliminate column i
        in row j using row i;
        save pivot element in A[j,i];
        j:=j+p
      end
  end
```

Algorithm 2

The parallel LU-factorization involves both calculation and communication, and the execution time using p processors can be modelled as follows, ignoring terms in lower orders of N,

$$T_{PLU} = \frac{2}{3}\gamma^2 N^3 T_F/p +$$

$$(N-1)d(T_0 + \frac{1}{2}\gamma N/B) \quad (4)$$

The average fraction of nonzero elements of a row of the sparse matrix is denoted by γ . The floating point execution time is denoted by T_F and the start-up time of communication by T_0 . B denotes communication bandwidth (words/sec) and d is the dimension of the hypercube parallel computer $(p=2^d)$.

The analogous execution time model for Algorithm 1 is

$$T_{SLU} = \frac{2}{3} \gamma^2 N^3 T_F$$

If the communication term of (4) could be ignored, the speed-up, $S_{LU} = T_{SLU}/T_{PLU}$, would be equal to p. Unfortunately, this is rarely the case since $T_0 \approx 24T_F$ on the iPSC. Therefore the opposite situation, where communication time dominates over computation time, is more likely to arise, especially when the matrix is very sparse ($\gamma \ll 1$). In this situation, speed-up decreases when more processors are applied (d increases).

When the coefficient matrix of a system of linear equations is factored into an LU product,

$$LUx = b$$

the solution is computed by a forward substitution, $y = L^{-1}b$, followed by a backward substitution, $x = U^{-1}y$. A parallel version of the forward substitution is outlined in the following algorithm running on processor k. L is stored in the lower half triangular part of A.

```
for i:=1 to N do
  begin
    {if row i is on processor k ..}
    if i in [k,k+p,k+2p,..] then
      begin
        y[i]:=b[i];
        broadcast y[i]
      end
    else receive y[i];
    {row j is the first row on
    processor k where j>i}
    j:=((i+p-k) \text{ div } p)*p+k;
    while j<=N do
      begin
        b[j]:=b[j]-L[j,i]*y[i];
        j:=j+p
      end
 end
```

Algorithm 3

The backward substitution is very similar and will not be shown. The execution time model of the complete parallel solution algorithm is as follows,

$$T_{PS} = 2\gamma N^2/p + 2(N-1)d(T_0 + 1/B)$$
 (5)

The parallel solution algorithm is dominated by communication cost to even larger extent than the LU-factorization. Because the number of broadcasts of the solution algorithm is twice the number of broadcasts of the factorization algorithm, the execution time of the solution algorithm may exceed the execution time of the factorization algorithm. This is most unsatisfactory since the complexity of solution is $O(N^2)$ while the complexity of factorization is $O(N^3)$.

A significant improvement is obtained by including forward substitution, Algorithm 3, in the LU-factorization Algorithm 2. This is actually the classical Gaussian elimination, and the improvement is obtained by including the y[i] values in the pivot rows which are broadcast. This way, half of the communications of the solution algorithm is saved. The full advantage of this is gained in the true Newton iteration version of (3). The advantage is less for the pseudo Newton iteration where the Gaussian elimination is only used in the first iteration where the Jacobian is computed and LU-factorized.

The execution time model (4) includes the factor 1/p to reflect the parallel work performed by p processors. Each processor is responsible for the elimination of N/p rows, but if the number of non-zero elements of a column is less than the number of processors $(\gamma N < p)$, some processors will be idle. Therefore the effective number of processors may be less than p. This phenomenon together with the relatively high cost of the start-up of a communication, T_0 , motivates a modification of Algorithm 2 into a block version.

The processor allocation of the rows is changed such that blocks of b consecutive rows are allocated in an interleaved scheme. Processor k will therefore hold rows (k-1)b+1, (k-1)b+2,..., kb, (k-1+p)b+1, (k-1+p)b+2,..., (k+p)b,.. Algorithm 2 is modified to receive and broadcast, respectively, blocks of b rows in stead of single rows. The number of communications is then reduced by a factor of b, and the amount of work between communi-

cations is increased by a factor of b.

The general step of the block LU-factorization on processor k can be described informally as follows. Receive a block of b pivot rows and perform elimination. The total time for this step is T_B . If processor k is going to supply the next block of pivot rows, the pivot rows are first applied to this block (total time T_b). Then elimination within the block of pivot rows is performed (total time T_T), and the block of pivot rows are broadcast (total time T_C). Finally, the elimination with the block of pivot rows last received is completed.

The execution time models of the elimination steps are as follows,

$$T_B = \frac{2}{3}\gamma^2 N^3 T_F/p \tag{6}$$

$$T_b = b\gamma^2 N^2 T_F \tag{7}$$

$$T_T = \frac{1}{2}(b-1)\gamma^2 N^2 T_F$$
 (8)

$$T_C = dT_0(N/b-1) + \frac{1}{2}d\gamma N^2/B$$
 (9)

The resulting execution time of the block version of the parallel LU-factorization algorithm is therefore,

$$T_{bLU} = T_B + T_b + T_T + T_C \tag{10}$$

The limitations of this rather crude model are discussed in Section 7.

The execution time of the solution algorithm T_{PS} given in (5) is composed of a computation and a communication term. The communication term is divided by the number of rows in a block b when blocking is introduced, and the computation term is essentially unchanged.

However, the blocking introduces a substantial amount of overhead in terms of buffer administration, index computation etc. which reduces the gain of blocking. The overhead is not related to the floating point operations of the solution algorithm in a simple way, and a detailed modelling is not attempted.

6 Processor Allocation

The allocation of node equations (matrix rows) to processors was discussed in the previous section. The basic principle is the row interleaved scheme where runs of p consecutive node equations (rows) are allocated to p different processors and p is the number of available processors. This principle is modified to the block row interleaved scheme where b consecutive rows are lumped together and treated as one in the allocation scheme.

The purpose of row interleaving is to balance load on the processors during the LU-factorization and solution. In this respect it is important that a run of p consecutive rows are allocated to p different processors. However, it is not important where the rows are allocated. In other words, two rows in a run of p rows are allocated to two different processors, and they can be interchanged freely. This freedom in allocating node equations (rows) within a run is exploited to reduce the number of duplicate function calculations mentioned in Section 4 and to improve the load balance during computation of nonlinear functions.

Based on the circuit description input to ESACAP, an $N \times N$ array M is constructed where M[i,j] contains the number of nonlinear two terminal elements between node i and node j. The matrix is symmetric and the diagonal M[i,i] contains the number of two terminal elements from node i to ground. If the node equations i and j both are on processor k, the nonlinear two terminal elements between nodes i and j only have to be evaluated once, i. e. only processor k has to evaluate these M[i,j] functions. If node equations i and j are on processors k and k, respectively, both these processors must compute the M[i,j] functions.

Let r denote the number of rows on a processor (r = N/p), and let the array R_k contain row numbers for the rows on processor k. The number of functions, F_k , to be computed by processor k, can be evaluated by the following algorithm,

```
for i:=1 to r do
  begin
  {add gross number of functions}
  for j:=1 to N do
    Fk:=Fk+M[Rk[i],j];
  {subtract number of functions
    counted twice}
  for j:=1 to i-1 do
    Fk:=Fk-M[Rk[i],Rk[j]]
end
```

Algorithm 4

The initial processor allocation is recorded and the following Monte Carlo algorithm is used to improve the allocation by interchanging rows within a run of p consecutive rows,

```
for run:=1 to r do
  for it:=1 to maxit do
  begin
    improvement:=true;
    while improvement do
      begin
        chose randomly two different
        rows, s and t from run;
        {rows s and t are on proces-
        sors ps and pt, respectively}
        evaluate the effect on Fps and
        Fpt of interchanging them;
        {Algorithm 4}
        improvement:=
        {reduced work load}
        ((Fps is not increased) and
         (Fpt is not increased)) or
        {improved load balance}
        ((Fps decreases) and Fpt<=Fps)
        ((Fpt decreases) and Fps<=Fpt);
        if improvement then
          interchange rows s and t
      end
  end
```

Algorithm 5

Algorithm 5 is repeated until no more improvements are obtained. When several consecutive rows are allocated to a processor as a block they are treated as one long row by the algorithm. The first version of the processor allocation algorithm was based on simulated annealing, but it turned out to be much more expensive and only marginally better than the simple Monte Carlo algorithm.

Tables 1-3 show the results obtained by the Monte Carlo reallocation algorithm applied to

functions	initial	reallocated
max F	94	66
$ar{F}$	63.2	61.7
σ	14.4	3.0

Table 1: Nonlinear functions per processor. Rows per block b = 1.

functions	initial	reallocated
max F	98	71
$ar{F}$	62.5	61.3
σ	17.1	6.7

Table 2: Nonlinear functions per processor. Rows per block b = 2.

functions	initial	reallocated
max F	92	78
$ar{F}$	61.4	61.0
σ	16.6	11.3

Table 3: Nonlinear functions per processor. Rows per block b = 3.

the model of the 4×4 bit multiplier. The total number of nonlinear functions of the 180 node equations modelling the multiplier is 1118. When the node equations are allocated to 32 processors, approximately 2000 functions must be computed because of the need for duplicate computation.

The main effect of the Monte Carlo algorithm is to reduce the maximum number of functions allocated to one processor (max F) and thus improve load balance. This is also clearly reflected by the standard deviation σ of the number of functions while the average number of functions \bar{F} to be computed by a processor is only reduced little.

With increased number of rows in a block, the freedom to reallocate is reduced, and thus the efficiency of the Monte Carlo algorithm. This is reflected by the maximum number of functions and by the standard deviation.

7 Results

The performance of the parallel implementation of ESACAP is evaluated using a 4×4 bit version of the multiplier described in Section 2. The input of the multiplier is a sequence of binary numbers, 0000×0000 , 0010×0010 , 1100×1100 , 1011×1011 , 0100×0100 and 1111×1111 . The duration of each digit is 20 nsec and the transition from one level to another takes 10 nsec. The total simulated time is 170 nsec.

				T = 7=		
_ p	b	T	F	F_1/F_p	T_1/T_p	
1	-	133,200	1118	-	-	•
8	3	29,870	231	4.84	4.46	
16	3	17,892	129	8.67	7.44	
32	2	12,124	71	15.75	11.0	

Table 4: Execution time and speed-up figures for parallel ESACAP

The main performance figures are given in Table 4 which lists the total execution time (T) in seconds in column 3 and the speed-up over one processor (T_1/T_p) in column 6. The Cube Manager and the sequential ESACAP was used for p=1 because one node processor with only 512KB memory is too small to hold the problem. At least 8 processors are required to simulate the multiplier which explains why Table 4 does not have entries for $2 \le p \le 4$.

The number of rows in a block (b) is chosen to minimize the execution time. F denotes the maximum number of nonlinear functions to be computed in any processor, and F_1/F_p is the ratio of nonlinear functions in the sequential version over maximum number of nonlinear functions on one processor in the parallel version. This ratio is seen to be strongly correlated with the speed-up, T_1/T_p . However, the discrepancy increases with increasing value of p since the solution of linear equations, which does not parallelize well, then becomes relatively more important (see Table 5).

The speed-up obtained for less than 32 processors is satisfactory. For 32 processors the problem which has 180 equations is too small to maintain a speed-up value of approximately p/2.

Table 5 shows a break down of the execu-

P	b	T_{F+J}	T_{LU}	T_S
8	1	26,912	1,594	2,140
8	3	26,854	1,545	1,210
16	1	14,579	1,451	2,568
16	3	14,752	1,488	1,350
32	1	8,316	1,480	3,015
32	2	8,432	1,389	1,841

Table 5: Break down of execution time figures for parallel ESACAP

p	b	γ	T_B	T_C	T_b	T_T	T_{bLU}	T
8	1	0.19	866	644	-	-	1,511	1,511
8	2	-	866	322	175	45	1,408	1,394
8	3	-	866	215	264	90	1,435	1,415
8	4	-	866	161	351	135	1,513	1,574
8	5	-	866	130	440	175	1,610	1,745
16	1	0.20	492	859	_	-	1,351	1,351
16	2	-	492	430	195	48	1,165	1,276
16	3	-	492	287	293	96	1,168	1,361
32	1	0.23	316	1,073	-	-	1,389	1,389
32	2		316	536	257	64	1,173	1,272

Table 6: Execution time break down of one LU-factorization given in msec. The times $T_B - T_{bLU}$ refer to formulas (6) - (10). T is measured execution time.

tion times for the parallel simulations given in Table 4. T_{F+J} denotes the time spent computing the nonlinear functions of f_n and F'_n as defined in connection with (3), i. e. the nonlinear functions of the node equations and the corresponding derivatives. T_{LU} and T_S denote the time spent doing LU-factorization and solution, respectively. The execution times of Table 5 do not quite add up to the execution times of Table 4 because the latter includes some additional overhead.

The influence of blocking rows is displayed in Table 5 which includes execution times for b=1 and for block sizes giving minimum execution times. The blocking reduces the freedom in the processor allocation algorithm, and this leads to a slight increase in T_{F+J} for p=16 and p=32 (cf. Tables 1-3).

The LU-factorization only benefits from blocking for p=32, and this phenomenon is probably due to better processor utilization. With 180 node equations, the average number of rows per processor is less than 6, and with an average density of the Jacobi matrix of 0.1 (different from γ which is a model parameter), several processors will be idle during an elimination stage if pivot rows are broadcast one by one.

The solution algorithm gains most from blocking although the expected reduction of 1/b is not quite obtained.

Table 6 shows execution times based on the model, formulas (6) - (10). The times are in milliseconds and refer to one LU-factorization. The parameter γ which is an average row density used in the execution time model, is es-

timated for b=1. Therefore $T_{bLU}=T$ for b=1. The increase in γ with increasing p reflects the decrease in processor utilization. The remaining parameters of the model are as follows: $N=180,\,T_F=50\mu sec,\,T_0=1.2msec$ and B=250words/msec. Because of the substantial overhead involved in the operations modelled by (7) and (8), an increased value of the floating point execution time is used, $T_F=75\mu sec$.

The execution time model is quite accurate for p=8 and $b \le 4$. For larger values of b, the blocking leads to poor load distribution which is not modelled. This probably also accounts for the less accurate values for p=16 and p=32. However, the model still explains satisfactorily why the execution time of LUfactorization is not reduced by the full amount of saving in communication (T_C) when blocking is introduced.

8 Conclusion

The DC and transient analysis part of the general circuit analysis program ESACAP was parallelized for the Intel iPSC with a modest effort relative to the size and complexity of the original sequential version.

A speed-up of the parallel version over a sequential version of approximately p/2 can be expected when certain conditions are fulfilled: the problem must be highly nonlinear (e. g. a digital electrical circuit) and the amount of work per processor must be adequate.

The present parallel LU-factorization is straightforward and not very efficient. This

part leaves room for substantial improvement, and the parallel version of ESACAP will be used in the future as a test vehicle in connection with research in sparse matrix parallel algorithms.

9 Acknowledgement

The programming involved in parallelizing ESACAP for the Intel iPSC was done by Bjarne Hansen. His effort was made possible through a grant from Danish Natural Science Research Council.

References

- [1] E. Lelarasmee, A. E. Ruehli and A. L. Sangiovanni-Vincentelli, "The waveform relaxation method for time-domain analysis of large scale integrated circuits", IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-1, pp. 131-145, 1982.
- [2] S. Mattisson, CONCISE a Concurrent Circuit Simulation Program, Ph. D. Thesis, Department of Applied Electronics, Lund Institute of Technology, Lund, 1986.
- [3] P. Stangerup, "ESACAP A PCimplemented general-purpose circuit simulator", IEEE Circuits and Devices Magazine, Vol. 4, No. 4, pp. 20-25, 1988.
- [4] C. W. Ho, A. E. Ruehli and P. A. Brenan, "The modified nodal approach to network analysis", IEEE Trans. Circuits and Systems, Vol. CAS-22, pp. 504-509, 1975.
- [5] M. J. D. Powell, "A hybrid method for nonlinear equations" in Numerical Methods for Non-linear Algebraic Equations, (Ed. P. Rabinowitz), Gordon&Breach, pp. 87-161, 1970.
- [6] N. H. E. Weste and K. Eshraghian, Principles of CMOS VLSI Design, Addison-Wesley, 1985.
- [7] C. W. Gear, "The potential for parallelism in ordinary differential equations", Report No. UIUCDCS-R-86-1246, Dept. Computer Science, University of Illinois at Urbana-Champaign, 1986.

[8] W. F. Tinney and J. W. Walker, "Direct solution of sparse network equations by optimally ordered triangular factorization", Proc. IEEE, Vol. 55, pp. 1801-1809, 1967.

Concurrent DASSL Applied to Dynamic Distillation Column Simulation

Anthony Skjellum Manfred Morari

California Institute of Technology Chemical Engineering; mail code 210-41 Pasadena, California 91125 e-mail: tony@perseus.ccsf.caltech.edu

Abstract

The accurate, high-speed solution of systems of ordinary differential-algebraic equations (DAE's) of low index is of great importance in chemical, electrical and other engineering disciplines. Petzold's Fortran-based DASSL is the most widely used sequential code for solving DAE's. We have devised and implemented a completely new C code, Concurrent DASSL, specifically for multicomputers and patterned on DASSL. In this work, we address the issues of data distribution and the performance of the overall algorithm, rather than just that of individual steps. Concurrent DASSL is designed as an open, application-independent environment below which linear algebra algorithms may be added in addition to standard support for dense and sparse algorithms. The user may furthermore attach explicit data interconversions between the main computational steps, or choose compromise distributions. A "problem formulator" (simulation layer) must be constructed above Concurrent DASSL, for any specific problem domain. We indicate performance for a particular chemical engineering application, a sequence of coupled distillation columns. Future efforts are cited in conclusion.

Introduction

In this paper, we discuss the design of a general-purpose integration system for ordinary differential-algebraic equations of low index, following up on our more preliminary discussion in [16]. The new solver, Concurrent DASSL, is a parallel, C-language implementation of the algorithm codified in Petzold's DASSL, a widely used Fortran-based solver for DAE's

[11,4], and based on a loosely synchronous model of communicating sequential processes [9]. Concurrent DASSL retains the same numerical properties as the sequential algorithm, but introduces important new degrees of freedom compared to it. We identify the main computational steps in the integration process; for each of these steps, we specify algorithms that have correctness independent of data distribution.

We cover the computational aspects of the major computational steps, and their data distribution preferences for highest performance. We indicate the properties of the concurrent sparse linear algebra as it relates to the rest of the calculation. We describe the proto-Cdyn simulation layer, a distillation-simulation-oriented Concurrent DASSL driver which, despite specificity, exposes important requirements for concurrent solution of ordinary DAE's; the ideas behind a template formulation for simulation are, for example, expressed.

We indicate formulation issues and specific features of the chemical engineering problem – dynamic distillation simulation. We indicate results for an example in this area, which demonstrates the feasibility of this method, but the need for additional future work, both on the sparse linear algebra, and on modifying the DASSL algorithm to reveal more concurrency, thereby amortizing the cost of linear algebra over more time steps in the algorithm.

Mathematical Formulation

We address the following initial-value problem consisting of combinations of N linear and nonlinear coupled, ordinary differential-algebraic equations over the interval $t \in [T_0, T_1]$:

IVP(**F**, **u**, **Z**₀,
$$[T_0, T_1]; N, P$$
):

$$\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t) = \mathbf{0}, \quad t \in [T_0, T_1], \tag{1}$$

$$\mathbf{Z}(t=T_0) \equiv \mathbf{Z}_0, \ \dot{\mathbf{Z}}(t=T_0) \equiv \dot{\mathbf{Z}}_0,$$

with unknown state vector $\mathbf{Z}(t) \in \Re^N$, known external inputs $\mathbf{u}(t) \in \Re^P$, where $\mathbf{F}(\bullet;t) \mapsto \Re^N$ and $\mathbf{Z}_0, \dot{\mathbf{Z}}_0 \in \Re^N$ are the given initial-value, derivative vectors, respectively. We will refer to Equation 1's deviation from 0 as the residuals or residual vector. Evaluating the residuals means computing $\mathbf{F}(\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}; t)$ ("model evaluation") for specified arguments $\mathbf{Z}, \dot{\mathbf{Z}}, \mathbf{u}$ and t.

DASSL's integration algorithm can be used to solve systems fully implicit in \mathbf{Z} and $\dot{\mathbf{Z}}$ and of index zero or one, and specially structured forms of index two (and higher) [4, Chapter 5], where the index is the minimum number of times that part or all of Equation 1 must be differentiated with respect to t in order to express $\dot{\mathbf{Z}}$ as a continuous function of \mathbf{Z} and t [4, page 17].

By substituting a finite-difference approximation $\mathcal{D}_i \mathbf{Z}$ for $\dot{\mathbf{Z}}$, we obtain:

$$\mathbf{F}_{\mathcal{D}}(\mathbf{Z}_i; \tau_i) \equiv \mathbf{F}(\mathbf{Z}_i, \mathcal{D}_i \mathbf{Z}_i, \mathbf{u}_i; t = \tau_i) = \mathbf{0},$$
 (2)

a set of (in general) nonlinear staticized equations. A sequence of Equation 2's will have to be solved, one at each discrete time $t = \tau_i$, $i = 1, 2, ..., M^1$, in the numerical approximation scheme; neither M nor the τ_i 's need be pre-determined. In DASSL, the variable step-size integration algorithm picks the τ_i 's as the integration progresses, based on its assessment of the local error. The discretization operator for $\dot{\mathbf{Z}}$, \mathcal{D} , varies during the numerical integration process and hence is subscripted as \mathcal{D}_i .

The usual way to solve an instance of the staticized equations, Equation 2, is via the familiar Newton-Raphson iterative method (yielding $\mathbf{Z}_i \equiv \mathbf{Z}_i^{\infty}$):

$$\mathbf{Z}_{i}^{k+1} = \mathbf{Z}_{i}^{k} - c \left\{ \nabla_{\mathbf{Z}} \mathbf{F}_{\mathcal{D}}(\mathbf{Z}_{i}^{m_{k}}; \tau_{i}) \right\}^{-1} \mathbf{F}_{\mathcal{D}}(\mathbf{Z}_{i}^{k}; \tau_{i}), k = 0, 1, \dots$$
(3)

given an initial, sufficiently good approximation \mathbf{Z}_i^0 . The classical method is recovered for $m_k = k$ and c = 1, whereas a modified (damped) Newton-Raphson method results for $m_k < k$ (respectively, c < 1). In the original DASSL algorithm and in Concurrent DASSL, the Jacobian $\nabla_{\mathbf{Z}} \mathbf{F}_{\mathcal{D}}(\mathbf{Z})$ is computed by finite differences rather than analytically; this departure leads in another sense to a modified Newton-Raphson method even though $m_k = k$ and c = 1 might always be satisfied. For termination, a limit $k < k^*$

is imposed; a further stopping criterion of the form $\|\mathbf{Z}_{i}^{k+1} - \mathbf{Z}_{i}^{k}\| < \epsilon$ is also incorporated (see Brenan *et al.* [4, pages 121-124]).

Following Brenan et al., the approximation $\mathcal{D}_i \mathbf{Z}$ is replaced by a BDF-generated linear approximation, $\alpha \mathbf{Z} + \beta$, and the Jacobian

$$\nabla_{\mathbf{Z}}\mathbf{F}(\mathbf{Z}, \alpha\mathbf{Z} + \beta, \mathbf{u}; t) = \frac{\partial \mathbf{F}}{\partial \mathbf{Z}} + \alpha \frac{\partial \mathbf{F}}{\partial \dot{\mathbf{Z}}}.$$
 (4)

From this approximation, we define $\mathbf{F}_{\alpha,\beta}(\mathbf{Z};\tau_i)$ in the intuitive way. We then consider Taylor's Theorem with remainder, from which we can easily express a forward finite-difference approximation for each Jacobian column (assuming sufficient smoothness of $\mathbf{F}_{\alpha,\beta}$) with a scaled difference of two residual vectors:

$$\mathbf{F}_{\alpha,\beta}(\mathbf{Z} + \delta_j; \tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i) = \left\{ \nabla_{\mathbf{Z}} \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i) \right\} \delta_j + O\left(||\delta_j||_2^2 \right)$$
(5)

By picking δ_j proportional to \mathbf{e}_j , the jth unit vector in the natural basis for \Re^N , namely $\delta_j = d_j \mathbf{e}_j$, Equation 5 yields a first-order-accurate approximation in d_j of the jth column of the Jacobian matrix:

$$\frac{\mathbf{F}_{\alpha,\beta}(\mathbf{Z} + \delta_j; \tau_i) - \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i)}{d_j} = \begin{cases} \nabla_{\mathbf{Z}} \mathbf{F}_{\alpha,\beta}(\mathbf{Z}; \tau_i) \rbrace \mathbf{e}_j + O(d_j), \\ j = 1, \dots, N \end{cases}$$
(6)

Each of these N Jacobian-column computations is independent and trivially parallelizable. It's well known, however, that for special structures such as banded and block n-diagonal matrices, and even for general sparse matrices, a single residual can be used to generate multiple Jacobian columns [4,8]. We discuss these issues as part of the concurrent formulation section below.

The solution of the Jacobian linear system of equations is required for each k-iteration, either through a direct (e.g., LU-factorization) or iterative (e.g., preconditioned-conjugate-gradient) method. The most advantageous solution approach depends on N as well as special mathematical properties and/or structure of the Jacobian matrix $\nabla_{\mathbf{Z}} \mathbf{F}_{\mathcal{D}}$. Together, the inner (linear equation solution) and outer (Newton-Raphson iteration) loops solve a single time point; the overall algorithm generates a sequence of solution points \mathbf{Z}_i , $i=0,1,\ldots,M$.

In the present work, we restrict our attention to direct, sparse linear algebra as described in [13], although future versions of Concurrent DASSL will support the iterative linear algebra approaches by Ashby, Lee, Brown, Hindmarsh et al. [3,5]. For the sparse

¹ and more at trial timepoints which are discarded by the integration algorithm.

LU factorization, the factors are stored and reused in the modified Newton scenario. Then, repeated use of the old Jacobian implies just a forward and back-solve step using the triangular factors L and U. Practically, we can use the Jacobian for up to about five steps [4]. The useful lifetime of a single Jacobian evidently depends somewhat strongly on details of the integration procedure [4].

proto-Cdyn - Simulation Layer

To use the Concurrent DASSL system on other than toy problems, a simulation layer must be constructed above it. The purpose of this layer is to accept a problem specification from within a specific problem domain, and formulate that specification for concurrent solution as a set of differential-algebraic equations, including any needed data. On one hand, such a layer could explicitly construct the subset of equations needed for each processor, generate the appropriate code representing the residual functions, and create a set of node programs for effecting the simulation. This is the most flexible approach, allowing the user to specify arbitrary nonlinear DAE's. It has the disadvantage of requiring a lot of compiling and linking for each run in which the problem is changed in any significant respect (including but not limited to data distribution), although with sophisticated tactics, parametric variations within equations could be permitted without re-compiling from scratch, and incremental linking could be supported.

We utilize a template-based approach here, as we do in the Waveform-Relaxation paradigm for concurrent dynamic simulation [15]. This is akin to the ASCEND II methodology utilized by Kuru and many others [10]. It is a compromise approach from the perspective of flexibility; interesting physical prototype subsystems are encapsulated into compiled code as templates. A template is a conceptual building block with states, non-states, parameters, inputs and outputs (see below). A general network made from instantiations of templates can be constructed at runtime without changing any executable code. User input specifies the number and type of each template, their interconnection pattern, and the initial value of systemic states and extraneous (non-state) variables, plus the value of adjustable parameters and more elaborate data, such as physical properties. The addition of templates requires new subroutines for the evaluation of the residuals of their associated DAE's, and also for interfacing to the remainder of the system (e.g., parsing of user input, interconnectivity issues). With suitable automated tools, this addition process can be made

straightforward to the user.

Importantly, the use of a template-based methodology does not imply a degradation in the numerical quality of the model equations or solution method used. We are not obliged to tear equations based on templates or groups of templates as is done in sequential-modular simulators [19,6], where "sequential" refers in this sense to the stepwise updating of equation subsets, without connection to the number of computers assigned to the problem solution.

Ideally, the simulation layer could be made universal. That is, a generic layer of high flexibility and structural elegance would be created once and for all (and without predilection for a specific computational engine). Thereafter, appropriate templates would be added to articulate the simulator for a given problem domain. This is certainly possible with high-quality simulators such as ASCEND II and Chemsim (a recent Fortranbased simulator driving DASSL and MA28 [2,11,7]). Even so, we have chosen to restrict our efforts to a more modest simulation layer, called proto-Cdyn, which can create arbitrary networks of coupled distillation columns. This restricted effort has required significant effort, and already allows us to explore many of the important issues of concurrent dynamic simulation. General-purpose simulators are for future consideration. They must address significant questions of user-interface in addition to concurrency-formulation issues.

In the next paragraphs, we describe the important features of proto-Cdyn. In doing so, we indicate important issues for any Concurrent DASSL driver.

Template Structure

A template is a prototype for a sequence of DAE's which can be used repeatedly in different instantiations. Normally, but not always, the template corresponds to some subsystem of a physical-model description of a system, like a tank or distillation tray. The key characteristics of a template are: the number of integration states it incorporates (typically fixed), the number of non-state variables it incorporates (typically fixed), its input and output connections to other templates, and external sources (forcing functions) and sinks. State variables participate in the overall DASSL integration process. Non-states are defined as variables which, given the states of a template alone, may be computed uniquely. They are essentially local tear variables. It is up to the template designer whether or not to use such local tear variables: They impact the numerical quality of the solution, in principle. Alternative formulations, where all variables of a template are treated as states, can be posed, and comparisons made. Because of the superlinear growth of linear algebra complexity, the introduction of extra integration states must be justified on the basis of numerical accuracy. Otherwise, they artificially slow down the problem solution, perhaps significantly. Non-states are extremely convenient, and practically useful; they appear in all the dynamic simulators we have come across.

The template state and non-state structure implies a two-phase residual computation. First, given a state Z, the non-states of each template are updated on a template-by-template basis. Then, given its states and non-states, inputs from other templates and external inputs, each template's residuals may be computed. In the sequential implementation, this poses no particular nuisances, other than two evaluation loops over all templates. However, in concurrent evaluation, a communication phase intervenes between non-state updates and residual updates. This communication phase transmits all states and non-states appearing as outputs of templates to their corresponding inputs at other templates. This transmission mechanism is considered further below under concurrent formulation.

Problem Preformulation

In general, the "optimal" ordering for the equations of a dynamic simulation will in general be too difficult to establish², because of the NP-hard issues involved in structure selection. However, many important heuristics can be applied, such as those that precedence order the nonlinear equations, and those that permute the Jacobian structure to a more nearly triangular or banded form [8]. For the proto-Cdyn simulator, we skirt these issues entirely, because it proves easy to arrange a network of columns to produce a "good structure" – a main block tri-diagonal Jacobian structure with off-block-diagonal structure for the intercolumn connections, simply by taking the distillation columns with their states in tray-by-tray, top-down (or bottom-up) order.

Given a set of DAE's, and an ordering for the equations and states (i.e., rows and columns of the Jacobian, respectively), we need to partition these equations between the multicomputer nodes, according to a two-dimensional process grid of shape PxQ = R. The partitioning of the equations forms, in main part, the so-called "concurrent database." This grid structure is illustrated in [13, Figure 2.]. In proto-Cdyn, we

utilize a single process grid for the entire Concurrent DASSL calculation. That is, we don't currently exploit the Concurrent DASSL feature which allows explicit transformations between the main calculational phases (see below). In each process column, the entire set of equations is to be reproduced, so that any process column can compute not only the entire residual vector for a prediction calculation, but also, any column of the Jacobian matrix.

A mapping between the global equations and local equations must be created. In the general case, it will be difficult to generate a closed-form expression for either the global-to-local mapping or its inverse (that also require < O(N) storage). At most, we will have on a hand a partial (or weak) inverse in each process, so that the corresponding global index of each local index will be available. Furthermore, in each node, a partial global-to-local list of indices associated with the given node will be stored in global sort order. Then, by binary search, a weak global-to-local mapping will be possible in each process. That is, each process will be able to identify if a global index resides within it, and the corresponding local index. A strong mapping for row (column) indices will require communication between all the processes in a process row (respectively, column). In the foregoing, we make the tacit assumption that is is an unreasonable practice to use storage proportional to the entire problem size N in each node, except if this unscalability can be removed cheaply when necessary for large problems.

The proto-Cdyn simulator works with templates of specific structure – each template is a form of a distillation tray and generates the same number of integration states. It therefore skirts the need for weak distributions. Consequently, the entire row mapping procedure can be accomplished using the closed-form general two-parameter distribution function family ξ described in [13], where the block size B is chosen as the number of integration states per template. The column mapping procedure is accomplished with the one-parameter distribution function family ζ also described in [13]. The effects of row and column degree-of-scattering are described in [13] with attention to linear algebra performance.

Concurrent Formulation

Overview

Next, we turn to Equation 1's (that is, IVP's) concurrent numerical solution via the DASSL algorithm. We cover the major computational steps in abstract, and we also describe the generic aspects of proto-Cdyn in

²Optimality per se hinges on what our objective is. If, for instance, we want minimum time for LU factorization, still the objective of minimum fill-in does not guarantee minimum time in a concurrent setting.

this connection. In the subsequent section, we discuss issues peculiar to the distillation simulation.

Broadly, the concurrent solution of IVP consists of three block operations: startup, dynamic simulation, and a cleanup phase. Significant concurrency is apparent only in the dynamic simulation phase. We will assume that the simulation interval requested generates enough work so that the startup and cleanup phases prove insignificant by comparison and consequently pose no serious Amdahl's-law bottleneck. Given this assumption, we can restrict our attention to a single step of IVP as illustrated schematically in Figure 0.

In the startup phase, a sequential host program interprets the user specification for the simulation. From this it generates the concurrent database: the templates and their mutual interconnections, data needed by particular templates, and a distribution of this information among the processes that are to participate. The processes are themselves spawned and fed their respective databases. Once they receive their input information, the processes re-build the data structures for interfacing with Concurrent DASSL, and for generating the residuals. Tolerances, and initial derivatives must be computed and/or estimated. Furthermore, in each process column, the processes must rendezvous to finalize their communication labeling for the transmission of states and non-states to be performed during the residual calculation. This provides the basis for a reactive, deadlock-free update procedure described below.

The cleanup phase basically retrieves appropriate state values and returns them to the host for propagation to the user. Cleanup may actually be interspersed intermittently with the actual dynamic simulation. It provides simple bookkeeping of the results of simulation and terminates the concurrent processes at the simulation's conclusion.

The dynamic simulation phase consists of repetitive prediction and correction steps, and marches in time. Each successful time step requires the solution of one or more instances of Equation 2 – additional timesteps that converge but fail to satisfy error tolerances, or fail to converge quickly enough, are necessarily discarded. In the next section, we cover the aspects of these operations in more detail, for a single step.

Single Integration Step

The Integration Computations of DASSL are a fixed leading-coefficient, variable-stepsize and order, backward-differentiation-formula (BDF) implicit integration scheme, described clearly in [4, Chapter 5] and

outlined in [11]. Concurrent DASSL faithfully implements this numerical method, with no significant differences. Test problems run with the DASSL Fortran code and the new C code (on one and multiple computers) certify this degree of compatibility.

The sequential time complexity of the integration computations is O(N), if considered separately from the residual calculation called in turn, which is also normally O(N) (see below). We pose these operations on a PxQ = R grid, where we assume that each process column can compute complete residual vectors. Each process column repeats the entire prediction operations: there is no speedup associated with Q > 1, and we replicate all DASSL BDF and predictor vectors in each process column. Taller, narrower grids are likely to provide the overall greatest speedup, though the residual calculation may saturate (and slow down again) because of excessive vertical communication requirements — It's definitely not true that the Rx1 shape is optimal in all cases.

The distribution of coefficients in the rows has no impact on the integration operations, and is dictated largely by the requirements of the residual calculation itself. In practical problems, the concurrent database cannot be reproduced in each process (cf., [18]), so a given process will only be able to compute some of the residuals. Furthermore, we may not have complete freedom in scattering these equations, because there will often be a tradeoff between the degree of scattering and the amount of communication needed to form the entire residual vector.

The amount of O(N) integration-computation work is not terribly large — there is consequently a non-trivial but not tremendous effort involved in the integration computations. (Residual computations dominate in many if not most circumstances.) Integration operations consist mainly of vector-vector operations not requiring any interprocess communication and, in addition, fixed startup costs. Operations include prediction of the solution at the time point, initiation and control of the Newton iteration that "corrects" the solution, convergence and error-tolerance checking, and so forth. For example, the approximation \mathcal{D}_i is chosen within this block using the BDF formulas. For these operations, each process column currently operates independently, and repetitively forms the results. Alternatively, each process column could stride with step Q, and row-combines could be used to propagate information across the columns [14]. This alternative would increase speed for sufficiently large problems, and can easily be implemented. However, because of load-imbalance in other stages of the calculation, we are convinced that including this type of synchronization could be an overall negative rather than positive to performance. This alternative will nevertheless be a future user-selectable option.

Included in these operations are a handful of norm operations, which constitute the main interprocess communication required by the integration computations step; norms are implemented concurrently via recursive doubling (combine) [17,14]. Actually, the weighted norm used by DASSL requires two recursive doubling operations, each combines a scalar: first to obtain the vector coefficient of maximum absolute value, then to sum the weighted norm itself. Each can be implemented as Q independent column combines, each producing the same repetitive result, or a single Q-striding norm, that takes advantage of the repetition of information, but utilizes two combines over the entire process grid. Both are supported in Concurrent DASSL, although the former is the default norm. As with the original DASSL, the norm function can be replaced, if desired.

Single Residuals are computed in prediction, and as needed during correction. Multiple residuals are computed when forming the finite-difference Jacobian. Single residuals are computed repetitively in each process column, whereas the multiple residuals of a Jacobian computation are computed uniquely in the process columns.

Here, we consider the single residual computation required by the integration computations just described. Given a state vector Z, and approximation for Z, we need to evaluate $\mathbf{F}(\mathbf{Z}, \mathbf{Z}, \tau_i) \equiv \mathbf{F}_{\mathcal{D}}(\mathbf{Z}, \tau_i)$. The exploitable concurrency available in this step is strictly a function of the model equations. As defined, there are N equations in this system, so we expect to use at best N computers for this step. Practically, there will be interprocess communication between the process rows, corresponding to the connectivity among the equations. This will place an upper limit on $P \leq K$ (the number of row processes) that can be used before the speed will again decrease: we can expect efficient speedup for this step provided that the cost of the interprocess communication is insignificant compared to the single-equation grain size. As estimated in [14], the granularity T_{comm}/T_{calc} for the Symult s2010 multicomputer is about fifty, so this implies about four hundred and fifty floating point operations per communication in order to achieve 90% concurrent efficiency in this phase.

Jacobian Computation There is evidently much more available concurrency in this computational step

than for the single residual and integration operations, since, for finite differencing, N independent residual computations are apparently required, each of which is a single-state perturbation of \mathbf{Z} . Based on our overview of the residual computation, we might naively expect to use $K \times N$ processes effectively; however, the simple perturbations can actually require much less model evaluation effort because of latency [8,10], which is directly a function of the sparsity structure of the model equations, Equation 1. In short, we can attain the same performance with much less than $K \times N$ processors.

In general, we'd like to consider the Jacobian computation on a rectangular grid. For this, we can consider using $P \times Q = R$ to accomplish the calculation. With a general grid shape, we exploit some concurrency in both the column evaluations and in the residual computations, with $T_{Jac,PxQ=R}$ the time for this step, $S_{Jac,PxQ=R}$ the corresponding speedup, $T_{res,P}$ the residual evaluation time with P row processes, and $S_{res,P}$ the apparent speedup compared to one row process:

$$\begin{array}{ll} T_{Jac,PxQ=R} & \approx & \lceil N/Q \rceil \times T_{res,P} \\ S_{Jac,PxQ=R} & \approx & \frac{N}{\lceil N/Q \rceil} \times S_{res,P} \end{array}$$

assuming no shortcuts are available as a result of latency. This timing is exemplified in the example below, which does not take advantage of latency.

There is additional work whenever the Jacobian structure is rebuilt for better numerical stability in the subsequent LU factorization (A-mode). Then, $O(N^2/PQ)$ work is involved in each process in the filling of the initial Jacobian. In the normal case, work proportional to the number of local non-zeroes plus fill elements is incurred in each process for re-filling the sparse Jacobian structure.

Exploitation of Latency has been considered in We currently the Concurrent DASSL framework. have experimental versions of two mechanisms, both of which are designed to work with the sparse-matrix structures associated with direct, sparse LU factorization (see [13]). The first is called "bandlike" Jacobian evaluation. For a banded Jacobian matrix of bandwidth b, only b residuals are needed to evaluate the Jacobian. This feature is incorporated into the original DASSL, along with a LINPACK banded solver. In Concurrent DASSL, collections of Jacobian columns are placed in each process column, according to the column data distribution, which thus far is picked solely to balance LU factorization and triangular-solve performance [13]. In each process column, there will be

"compatible" columns that can be evaluated using a single, composite perturbation. Identification of these compatible columns is accomplished by checks on the bandwidth overlap condition. Columns that possess off-band structure are stricken from the list and evaluated separately. Presumably, a heuristic algorithm could be employed further to increase the size of the compatible sets, but this is yet to be implemented. The same algorithm "greedy" algorithm of Curtis et al. used for the sequential reduction of Jacobian computation effort would be applied independently to each process column (see comments by [8, Section 12.3]). Then, clearly, the column distribution effects the performance of the Jacobian computation, and the linearalgebra performance can no longer be viewed so readily in isolation.

We have also devised a "blocklike" format, which will be applied to block n-diagonal matrices that include some off-block entries as well. Optimally, fewer residual computations will be needed than for the banded case. The same column-by-column compatible sets will be created, and the Curtis algorithm can also be applied. Hopefully, because of the less restrictive compatibility requirement, the "blocklike" case will produce higher concurrent speedups than that attained using the conservative bandlike assumption for Jacobians possessing blocklike structure. Comparative results will be presented in a future paper.

The LU Factorization Following the philosophy of Harwell's MA28, we have interfaced a new concurrent sparse solver to Concurrent DASSL, the details of which are quoted elsewhere in this proceedings [13]. In short, there is a two-step factorization procedure: A-mode, which chooses stable pivots according to a user-specified function, and builds the sparse data structures dynamically; and B-mode, which re-uses the data structures and pivot sequence on a similar matrix, but monitors stability with a growth-factor test. A-mode is repeated whenever necessary to avoid instability. We expect sub-cubic time complexity and sub-quadratic space complexity in N for the sparse solver. We attain acceptable factorization speedups for systems that are not narrow banded, and of sufficient size. We intend to incorporate multiple pivoting heuristic stategies, following [1], further to improve performance of future versions of the solver. This may also contribute to better performance of the triangular solves.

Forward- and Back-solving Steps take the factored form

$$\mathsf{P}_R A \mathsf{P}_C^T = \hat{L} \hat{U},$$

with \hat{L} unit lower-triangular, \hat{U} upper-triangular, and permutation matrices P_R , P_C , and solve Ax = b, using the implicit pivoting approach described in [13]. Sequentially, the triangular solves each require work proportional to the number of entries in the respective triangular factor, including fill-in. We have yet to find an example of sufficient size for which we actually attain speedup for these operations, at least for the sparse case. At most, we try to prevent these operations from becoming competitive in cost to the B-mode factorization; we detail these efforts in [13]. In brief, the optimum grid shape for the triangular solves has Q = 1, and P somewhat reduced than what we can use in all the other steps. As stated, P small seems better thus far, though for many examples, the increasing overhead as a function of increasing P is not unacceptable (see [13] and the example below).

Residual Communication is an important aspect of the proto-Cdyn layer. As indicated in the startupphase discussion, the members of a process column initially share information about the groups of states and non-states they will exchange during a residual computation. For residual communication, a reactive transmission mechanism is employed, to avoid deadlocks. Each process transmits its next group of states to the appropriate process and then looks for any receipt of state information. Along with the state values are indices that directly drive the destinations for these values. This index information is shared during the startup phase and allows the messages to drive the operation. Through non-blocking receives, this procedure avoids problems of transmission ordering. Regardless of the template structure, at most one send and receive is needed between any pair of column processes.

Chemical Engineering Example

The algorithms and formalism needed to run this example amount to about 70,000 lines of C code including the simulation layer, *Concurrent DASSL*, the linear algebra packages, and support functions [14,13,12].

In this simulation, we consider seven distillation columns arranged in a tree-sequence [12], working on the distillation of eight alcohols: methanol, ethanol, propan-1-ol, propan-2-ol, butan-1-ol, 2-methyl propan-1-ol, butan-2-ol, and 2-methyl propan-2-ol. Each column has 143 trays. Each tray is initialized to a non-steady condition, and the system is relaxed to the steady state governed by a single feed stream to the first column in the sequence. This setup

generates suitable dynamic activity for illustrating the cost of a single "transient" integration step.

We note the performance in Table 0. Because we have not exploited latency in the Jacobian computation, this calculation is quite expensive, as seen for the sequential times on a Sun 3/260 depicted there. (The timing for the Sun 3/260 is quite comparable to a single Symult s2010 node and was lightly loaded during this test run.) As expected, Jacobian calculations speedup efficiently, and we are able to get approximately a speedup of 100 for this step using 128 nodes. The A-mode linear algebra also speeds up significantly. The B-mode factorization speeds up negligibly and quickly slows down again for more than 16 nodes. Likewise, the triangular solves are significantly slower than the sequential time. It should be noted that B-mode reflects two orders of magnitude speed improvement over A-mode. This reflects the fact that we are seeing almost linear time complexity in B-mode, since this example has a narrow block tri-diagonal Jacobian with too little off-diagonal coupling to generate much fill-in. It seems hard to imagine speeding up B-mode for such an example, unless we can exploit multiple pivots. We expect multiple-pivot heuristics to do reasonably well for this case, because of its narrow structure, and nearly block tri-diagonal structure. We have used Wilson Equation Vapor-Liquid Equilibrium with the Antoine Vapor equation. We have found that the thermodynamic calculations were much less demanding than we expected, with bubble-point computations requiring " $1+\epsilon$ " iterations to converge. Consequently, there was not the greater weight of Jacobian calculations we expected beforehand. Our model assumes constant pressure, and no enthalpy balances. We include no flow dynamics and include liquid and vapor flows as states, because of the possibility of feedbacks.

Were we to utilize latency in the Jacobian calculation, we could reduce the sequential time by a factor of about 100. This improvement would also carry through to the concurrent times for Jacobian solution. At that ratio, Jacobian computation to B-mode factorization has a sequential ratio of about 10:1. As is, we achieve legitimate speedups of about five. We expect to improve these results using the ideas quoted elsewhere here and in [13].

From a modeling point-of-view, two things are important to note. First, the introduction of more non-ideal thermodynamics would improve speedup, because these calculations fall within the Jacobian computation phase and Single-Residual Computation. Furthermore, the introduction of a more realistic model will likewise bear on concurrency, and likely im-

prove it. For example, introducing flow dynamics, enthalpy balances and vapor holdups makes the model more difficult to solve numerically (higher index). It also increases the chance for a wide range of step-sizes, and the possible need for additional A-mode factorizations to maintain stability in the integration process. Such operations are more costly, but also have a higher speedup. Furthermore, the more complex models will be less likely to have near diagonal dominance; consequently more pivoting is to be expected, again increasing the chance for overall speedup compared to the sequential case. Mainly, we plan to consider the Waveform-Relaxation approach more heavily, and also to consider new classes of dynamic distillation simulations with Concurrent DASSL [12].

Conclusions

We have developed a high-quality concurrent code, Concurrent DASSL, for the solution of ordinary a Terential-algebraic equations of low index. code, together with appropriate linear algebra and simulation layers, allows us to explore the achievable concurrent performance of non-trivial problems. In chemical engineering, we have applied it thus far to a reasonably large, simple model of coupled distillation columns. We are able to solve this large problem, which is quite demanding on even a large mainframe because of huge memory requirements and non-trivial computational requirements; the speedups achieved thus far are legitimately at least five, when compared to an efficient sequential implementation. This illustrates the need for improvements to the linear algebra code, which are feasible because sparse matrices will admit multiple pivots heuristically. It also illustrates the need to consider hidden sources of additional timelike concurrency in Concurrent DASSL, perhaps allowing multiple right-hand sides to be attacked simultaneously by the linear algebra codes, and amortizing their cost more efficiently. Furthermore, the performance points up the need for detailed research into the novel numerical techniques, such as Waveform Relaxation, which we have begun to do as well [15].

Acknowledgements

The first author acknowledges the kind assistance and helpful cooperation of Lionel F. Laroche and Henrik W. Andersen in the area of dynamic simulation for chemical process flowsheets. We have spent many hours together over the last twenty months in the discussion of design goals, features, algorithms, on realizations, post-mortems and re-designs, and in over-

Table 0. Order 9009 Dynamic Simulation Data								
		(time in seconds)						
Grid Shape	Jacobian	A-mode	B-mode	Back-Solve	Solve			
1x1	64672.2	5089.96	61.82	2.5	4.7			
8x1	6870.82	1024.41	47.827	15.619	30.825			
16x1	3505.13	547.625	52.402	19.937	39.491			
32x1	1829.93	316.544	56.713	24.383	47.692			
64x1	1060.40	219.148	77.302	39.942	59.553			
32x4	491.526	181.082	71.482	57.049	101.994			
64x2	520.029	161.052	82.696	46.013	86.935			
128x1	608.946	170.022	90.905	37.498	67.982			

Key single-step calculation times with the 1x1 case run an unloaded Sun 3/260 (similar performance-wise to a single Symult s2010 node) for comparison. The Jacobian rows were distributed in block-linear form, with B=9, reflecting the distillation-tray structure. The Jacobian columns were scattered. This is an seven column simulation of eight alcohols, with a total of 1,001 trays. See [13] for more on data distributions.

coming the stumbling blocks in our respective simulation codes. Thanks also to Prof. A. W. Westerberg of CMU, who offered helpful suggestions when he visited Caltech in 1989.

Thanks to Drs. K. E. Brenan, S. L. Campbell and Linda Petzold, for sharing advance drafts of their monograph Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, which proved very helpful in the creation of Concurrent DASSL.

The first author acknowledges partial support under DOE grants DE-FG03-85ER25009 and DE-AC03-85ER40050.

Concurrent DASSL was developed using machine resources made available by the Caltech Computer Science sub-Micron System Architectures Project and the Caltech Concurrent Supercomputer Facilities (CCSF).

References

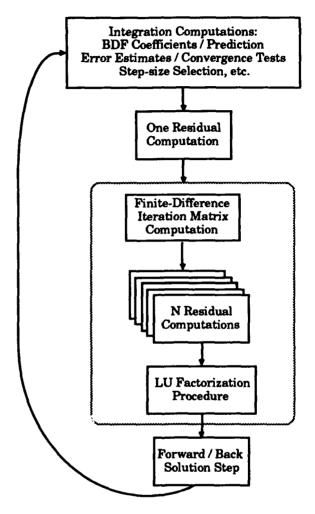
- [1] G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Computing*, 11:201-221, 1989.
- [2] H. W. Andersen and L. F. Laroche, 1988-1990.
 Private Communications on Chemsim.
- [3] S. Ashby, 1990. Private Communication on

Iterative DASSL.

- [4] K. E. Brenan, S. L. Campbell, and L. R. Petzold. Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations. North Holland Elsevier, 1989.
- [5] P. N. Brown and A. C. Hindmarsh. Reduced storage matrix methods in stiff ODE systems. J. Appl. Math. & Comp., (to appear).
- [6] W. J. Cook. A modular dynamic simulator for distillation systems. Master's thesis, Case Western Reserve University, 1980. Chemical Engineering.
- [7] I. S. Duff. MA28 a set of fortran subroutines for sparse unsymmetric linear equations. Technical Report R8730, AERE, HMSO, London, 1977.
- [8] I. S. Duff, A. M. Erisman, and J. K. Reid. Direct Methods for Sparse Matrices. Oxford University Press, 1986.
- [9] C. A. R. Hoare. Communicating sequential processes. CACM, 21(8):666-677, August 1978.
- [10] S. Kuru. Dynamic Simulation with an Equation Based Flowsheeting System. PhD thesis, Carnegie Mellon University, 1981. Chemical Engineering Department.

- [11] L. R. Petzold. DASSL: Differential algebraic system solver. Technical Report Category #D2A2, Sandia National Laboratories — Livermore, 1983.
- [12] A. Skjellum. Concurrent Dynamic Simulation: Multicomputer Algorithms Research Applied to Differential-Algebraic Process Systems in Chemical Engineering. PhD thesis, California Institute of Technology, May 1990. Chemical Engineering.
- [13] A. Skjellum and A. P. Leung. LU factorization of sparse, unsymmetric jacobian matrices on multicomputers: Experience, Strategies, Performance. In Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5). in press, April 1990.
- [14] A. Skjellum and A. P. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5). in press, April 1990.
- [15] A. Skjellum, M. Morari, and S. Mattisson. Waveform Relaxation for Concurrent Dynamic Simulation of Distillation Columns. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3), pages 1062– 1071. ACM Press, January 1988.
- [16] A. Skjellum, M. Morari, S. Mattisson, and L. Peterson. Concurrent DASSL: Structure, Application, and Performance. In Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications (HCCA4), pages 1321-1328. Golden Gate Enterprises, March 1989. Simulation Minisymposium.
- [17] H. S. Stone. High-Performance Computer Architecture. Addison-Wesley, 1987.
- [18] E. F. Van de Velde and J. Lorenz. Adaptive data distribution for concurrent continuation. Technical Report CRPC-89-4, California Institute of Technology, 1989. Caltech/Rice Center for Research in Parallel Computation.
- [19] A. W. Westerberg, H. P. Hutchison, R. L. Motard, and P. Winter. *Process flowsheeting*. Cambridge University Press, 1979.

Figure 0. Major computational blocks of a Single Integration Step.



A single step in the integration begins with a number of BDF-related computations, including the solution "prediction" step. Then, "correction" is achieved through Newton iteration steps, each involving a Jacobian computation, and linear-system solution (LU factorization plus forward/back-solves). The computation of the Jacobian in turn relies upon multiple independent residual calculations, as shown. The three items enclosed in the dashed oval (Jacobian computation (through at-most N Residual computations), and LU factorization) are, in practice, computed less often than the others — the old Jacobian matrix is used in the iteration loop until convergence slows intolerably.

CONVERGENCE AND CIRCUIT PARTITIONING ASPECTS FOR WAVEFORM RELAXATION

Ulla Miekkala Olavi Nevanlinna

Helsinki University of Technology Institute of Mathematics 02150 Espoo, Finland Albert Ruehli

IBM T. J. Watson Research Center Yorktown Heights NY 10598, U.S.A

ABSTRACT

This paper gives a mathematical investigation of the convergence properties of a model problem which, at first sight, seems to be unsuitable for waveform relaxation. The model circuit represents a limiting case for capacitive coupling where the capacitances to ground are zero. We show that the WR approach converges. Since the convergence is generally slow we discuss appropriate techniques for accelerating convergence.

1. INTRODUCTION

A substantial speed-up can be achieved in the analysis of large circuits by using the waveform relaxation (WR) method [4] rather than the conventional direct, incremental time (IT) methods. Another speed improvement can be obtained by applying the WR method to parallel processors since the approach is based on partitioning the circuit into small subcircuits which are assigned to the different processors. Hence, a central problem in the approach is the partitioning into the "best" possible subcircuits. Our experience has been that the gain of the WR over the IT method is a function of the partitioning.

In this short paper, we consider convergence for linear model circuits. For any RC-circuit, independently whether the index is one or two, geometric convergence can be proved if the partitioning can be performed across the resistors only [3]. However, this restriction will in many cases lead to large subcircuits. This is especially the case for MOSFET transistor circuits where the gate to drain capacitance plays a key role in the partitioning. Partitioning must be performed across capacitances in this case since partitioning across resistors only would lead to unduly large subcircuits. It is essential in some

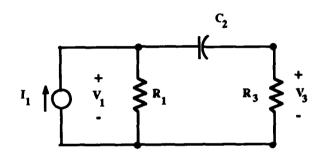


Figure 1.

cases that the partitioning is performed even for situations where the convergence cannot be achieved in only a few iterations. The connections among the subcircuits may simply lead to very large subcircuits. It is in most cases desirable to partition into subcircuits of a similar size. Non-uniform scheduling schemes, like the ϵ -scheduling, can take advantage of partitioning schemes with widely varying convergence rates at the subcircuit level [1].

Here we are investigating a very interesting model circuit shown in Figure 1. The iteration becomes

$$\dot{v}_{1}^{n} + \frac{1}{R_{1}C_{2}}v_{1}^{n} = \frac{I_{1}}{C_{2}} + \dot{v}_{3}^{n-1}$$

$$\dot{v}_{3}^{n} + \frac{1}{R_{3}C_{2}}v_{3}^{n} = \dot{v}_{1}^{n-1}$$
(1.1)

It is a limiting case, where a coupling capacitor is present, while the capacitors to ground are missing. This example represents a worst case situation which is somewhat non physical since each node has a capacitance to ground in a VLSI environment. Earlier convergence proofs show that the WR approach converges, if we partition this circuit into two subcircuits across the capacitor, provided that a capacitor to ground is present [4], [10].

Intuitively, one assumes that the absence of grounded capacitors will prevent convergence since the nodes are coupled together when $\xi \to \infty$. Here, we show that the WR solution for the circuit in Fig. 1 will indeed converge but that the speed of convergence is quite slow. This is included in Theorem 1 in Section 2. The result shows that the speed of convergence depends on the number of correct derivatives at t=0. In Section 3 we discuss the discretized iteration. One can show that for a fixed time step h the convergence is geometric, ie of the form ρ^n with ρ of the form 1/(1+ch). For the second order BDF-formula this is obtained if the coupling derivatives are read through a "filter". In Section 4 we shortly mention some possibilities for accelerating the convergence.

2. CONVERGENCE

The model problem (1.1) - without iteration index n - is an index one differential-algebraic equation (DAE). This can be seen by adding the two equations: we get an algebraic equation for the voltages. This means in particular that only one of the initial values can be freely chosen.

The iteration in (1.1) is of the form

$$\dot{x}_1^n + \lambda_1 x_1^n = \dot{x}_2^{n-1} + f_1
\dot{x}_2^n + \lambda_2 x_2^n = \dot{x}_1^{n-1} + f_2.$$
(2.1)

This is now an ODE system for (x_1^n, x_2^n) , but unless the initial values satisfy the extra restriction we cannot expect convergence.

If we denote by y^n the iteration error $y^n = x - x^n$, then we may assume $y^n(0) = 0$ for all n. Thus

$$\dot{y}_1^n + \lambda_1 y_1^n = \dot{y}_2^{n-1}
\dot{y}_2^n + \lambda_2 y_2^n = \dot{y}_1^{n-1}.$$
(2.2)

Taking the Laplace transform of (2.2) yields $(\tilde{y}(z) = \int_0^\infty e^{-zt} y(t) dt)$

$$\tilde{y}^{n}(z) = K(z)\tilde{y}^{n-1}(z),$$
 (2.3)

where

$$K(z) = \begin{pmatrix} 0 & \frac{s}{s + \lambda_1} \\ \frac{s}{s + \lambda_2} & 0 \end{pmatrix}. \tag{2.4}$$

According to the basic approach [5], [6] one then looks at the spectral radius of K(z) along $z = i\xi$:

$$\rho(K(i\xi)) = \left\{ \frac{\xi^2}{\lambda_1^2 + \xi^2} \frac{\xi^2}{\lambda_2^2 + \xi^2} \right\}^{1/4}.$$

In particular $\rho(K(i\xi)) < 1$ for $|\xi| < \infty$ but near infinity $\lim_{|\xi| \to \infty} \rho(K(i\xi)) = 1$. This means that the process

does not converge geometrically, say, in L_2 . However, as $\rho(K(i\xi)) < 1$ for all $|\xi| < \infty$ there is still convergence, slower and such that it depends on the smoothness of the initial error. As large values of $|\xi|$ correspond to fast time scales we may expect the convergence speed to depend on the smoothness of the initial error: the smoother the initial error the faster the convergence. We shall make this precise.

Consider the following iteration

$$R\dot{y}^{n} + My^{n} = S\dot{y}^{n-1} + Ny^{n-1}$$
 (2.5)

with y^0 given and $y^n(0) = 0$ for all n. Here R, M, S and N are square matrices such that

$$zR + M$$
 is nonsingular for $Rez > 0$. (2.6)

Then the symbol of the iteration (2.5)

$$K(z) = (zR + M)^{-1}(zS + N)$$

is an analytic matrix-valued function in Re z > 0. We make the following model assumption:

 \exists nonnegative constants C, a, b with a < 1 such that for all n

$$||K(i\xi)^n||^2 \le C^2 \left(\frac{a^2 + b^2 \xi^2}{1 + b^2 \xi^2}\right)^n,$$
 (2.7)

where || || denotes the matrix norm induced by the usual Euclidean length. In the example we have N=0 and (2.7) holds with a=0.

Next we need the Sobolev norms:

 $u \in H^{\bullet}$ iff

$$||u||_{\sigma} := \left\{ \frac{1}{2\pi} \int (1+\xi^2)^{\sigma} ||\tilde{u}(i\xi)||^2 d\xi \right\}^{1/2} < \infty.$$
(2.8)

Observe that $H^0 = L_2$.

From (2.5) and the definition of the symbol K(z) we see that

$$\tilde{y}^{n}(z) = K(z)^{n} \tilde{y}^{0}(z)$$

and by Parseval's identity

$$||y^{n}||_{0}^{2} = \frac{1}{2\pi} \int ||K(i\xi)^{n} \tilde{y}^{0}(i\xi)||^{2} d\xi.$$
 (2.9)

Now we make the smoothness assumption.

$$y^0 \in H^s$$
, for some $s > 0$. (2.10)

Using (2.6) and (2.7) in (2.8) we have

$$\begin{aligned} &||y^{n}||_{0}^{2} \\ &\leq \frac{1}{2\pi} \int ||K(i\xi)^{n}||^{2} (1+\xi^{2})^{-s} (1+\xi^{2})^{s} ||\tilde{y}^{0}(i\xi)||^{2} d\xi \\ &\leq \Psi^{2}(n) ||y^{0}||_{s}^{2}, \end{aligned}$$

where

$$\Psi^{2}(n) := \sup_{\xi} C^{2} \left(\frac{a^{2} + b^{2} \xi^{2}}{1 + b^{2} \xi^{2}} \right)^{n} (1 + \xi^{2})^{-s}.$$

A simple calculation yields

$$\Psi(n) = \mathcal{O}((\frac{1}{n})^{s/2}).$$

Theorem 1 Under the model assumptions (2.6), (2.7) and under the smoothness assumption (2.10) we have

$$||y^n||_0 = \mathcal{O}((\frac{1}{n})^{s/2})||y^0||_s.$$
 (2.11)

The actual exponent s in (2.10) depends strongly on the preparation of initial guess at t = 0. If only $y^0(0) = 0$ is assumed, then by partial integration we have

$$\bar{y}^{0}(z) = \frac{1}{z} \int e^{-xt} \dot{y}^{0}(t) dt.$$

If we set $z = \alpha + i\xi$ then this implies as $\alpha > 0$

$$|\bar{y}^0(z)| \leq \frac{\operatorname{Const}}{|z|\alpha}.$$

If $\tilde{y}^0(z)$ is analytic at infinity then

$$\left|\tilde{y}^0(z)\right|=\mathcal{O}(\frac{1}{|z|^2}),\;|z|\to\infty.$$

If we, however, prepare the initial guess so that

$$D^{j}y^{0}(0) = 0$$
 for $j = 0, ..., l-1$, (2.12)

then, performing l partial integrations and assuming as before that $\tilde{y}^0(z)$ is analytic at infinity, we have

$$|\tilde{y}^{0}(z)| = \mathcal{O}(\frac{1}{|z|^{l+1}}).$$
 (2.13)

Thus, integrating this along $z = i\xi$ (or along $z = \alpha + i\xi$ with a moderate $\alpha > 0$ if needed) we obtain

$$y^0 \in H^s$$
 for all $s < l + \frac{1}{2}$.

On the other hand, think y^0 to be given on the whole R and vanishing identically for $t \le 0$. Then it is clear that the continuity at origin shows up in the Sobolev exponent provided that the initial error is otherwise smooth. Recall that if $y^0 \in H^s$, then by the Sobolev embedding lemma y^0 has continuous derivatives up to $\lfloor s - \frac{1}{2} \rfloor$. Therefore, if (2.12) holds but $D^l y^0(0) \ne 0$ then $\lfloor s - \frac{1}{2} \rfloor \le l - 1$, and necessarily $s < l + \frac{1}{2}$.

Although Theorem 1 is quite sharp as such, the smoothness assumption in form of Sobolev norm $||y^0||_s$ does not contain information on where y^0 is spectrally large. As typically we would expect (2.12) only to hold with l=1, Theorem 1 only says that eventually the convergence is likely to be of the form $\frac{1}{s^r}$ with $r\sim 3/4$. To capture the decay in the early sweeps, we can look first again the example (2.1). For simplicity, let $\lambda_1=\lambda_2=1$, $x_2^0\equiv f_2\equiv 0$ and $x_1^0\equiv f_1$. Then, for the iteration error, we have

$$\tilde{y}_1^0(z) = -\tilde{y}_2^0(z) = -\frac{z}{2z+1}\tilde{f}_1(z)$$

If now, e.g. $f_1(t) = C(1 - e^{-\gamma t})$, then

$$\tilde{y}_1^0(z) = -C \frac{\gamma}{(2z+1)(z+\gamma)}.$$

Notice that $|\tilde{y}_1^0(z)| \sim C$ for small |z| and for large |z| $|\tilde{y}_1^0(z)| \sim \frac{C\gamma/2}{|z|^2}$.

We model the general case in the same way

$$|\tilde{y}^{0}(i\xi)| \le \min\{1, \frac{\gamma}{|\xi|^2}\},$$
 (2.14)

and again consider the iteration (2.5). Suppose that we would like to stop the iteration when $||y^n||_0 \le \epsilon$. Of course, we expect to see that n depends on $1/\epsilon$ and that for small γ we have rapid convergence. We present a simple-minded approach which shows this qualitatively, while for each special case the computation should be carried out in more detail. In estimating $||y^n||_0^2$ we break the integral into two parts:

$$I_{1} = \frac{1}{2\pi} \int_{|\xi| \ge T} |\bar{y}^{n}(i\xi)|^{2} d\xi,$$

$$I_{2} = \frac{1}{2\pi} \int_{|\xi| \le T} |\bar{y}^{n}(i\xi)|^{2} d\xi.$$

For I_1 we require $I_1 \leq \frac{C^2}{2\pi} \int \frac{\gamma^2}{\xi^4} d\xi \sim \epsilon^2$ which is the case if $T := \left(\frac{C\gamma}{\epsilon}\right)^{2/3}$. Now consider all other constants to be fixed (i.e. C, a, b) and think ϵ and γ as variables. We want $I_2 \sim \epsilon^2$, hence we approximate

$$I_2 \leq \frac{C^2}{2\pi} T \left(\frac{a^2 + b^2 T^2}{1 + b^2 T^2} \right)^n \sim \epsilon^2.$$

Here we have a geometric rate and accuracy level ϵ is reached with $n \sim \log(1/\epsilon)$ sweeps (assuming the initial accuracy level 1) provided T^2 stays bounded, i.e. $\gamma/\epsilon \leq \text{const.}$ We summarize

Theorem 2 Assume the model assumptions (2.6), (2.7) and initial error in the form (2.14). There exist constants c_1 and c_2 only depending on C, a, b in (2.7) such that for all small ϵ and γ with $\gamma/\epsilon \le c_1$ the accuracy requirement $||y^n||_0 \le \epsilon$ is reached with $n \sim c_2 \log(1/\epsilon)$ sweeps.

3. DISCRETIZATION

We consider the time discretization of (2.1) with constant time step h and multistep (k-step) method defined through its generating polynomials $a(\zeta) = \sum_{0}^{k} \alpha_{j} \zeta^{j}$ and $b(\zeta) = \sum_{0}^{k} \alpha_{j} \zeta^{j}$. There are several possibilities to discretize the right hand side of the equations. Since it contains derivative terms one could use simply the method defined by $a(\zeta)$. On the other hand, it can be treated as a source term since it is known from previous iteration. As we will see, discretization using a multiple of time step h and thus having a "filtering" effect is particularly interesting in our model problem. We denote the discretization of the RHS here defined by $\bar{a}(\zeta) = \sum_{l=0}^{k} \bar{\alpha}_{l} \zeta^{j}$ where l_{0} may be positive so that \bar{a} uses more steps than k. Equations for the iteration error become

$$\sum_{0}^{k} \alpha_{j} y_{1 j+\nu}^{n} + h \sum_{0}^{k} \beta_{j} y_{1 j+\nu}^{n} = \sum_{-l_{0}}^{k} \bar{\alpha}_{j} y_{2 j+\nu}^{n-1}$$

$$\sum_{0}^{k} \alpha_{j} y_{2 j+\nu}^{n} + h \sum_{0}^{k} \beta_{j} y_{2 j+\nu}^{n} = \sum_{-l_{0}}^{k} \bar{\alpha}_{j} y_{1 j+\nu}^{n-1}$$

$$\nu = 0, 1, \dots \quad y_{1,0}^{n} = y_{2,0}^{n} = 0.$$
(3.1)

In l_2 -space (3.1) can be analyzed by using the ζ -transformation (discrete Laplace transformation) because of Parseval's identity. With similar derivation as in [7] we obtain

$$(a(\zeta) + hb(\zeta)\lambda_1)\hat{y}_1^n(\zeta) = \bar{a}(\zeta)\hat{y}_2^{n-1}(\zeta)$$

$$(a(\zeta) + hb(\zeta)\lambda_2)\hat{y}_2^n(\zeta) = \bar{a}(\zeta)\hat{y}_1^{n-1}(\zeta)$$
(3.2)

where $\hat{y}(\zeta) = \sum_{j=0}^{\infty} y_j \zeta^{-j}$ for the sequence $\{y_j\}_0^{\infty}$. As

$$\hat{y}_1^n = \frac{\bar{a}}{a + hb\lambda_1} \frac{\bar{a}}{a + hb\lambda_2} \hat{y}_1^{n-2},$$

it is clearly sufficient to study the simpler (although nonphysical) model iteration

$$\dot{y}^n + y^n = \dot{y}^{n-1} \,. \tag{3.3}$$

After discretization and ζ -transformation we namely get

$$(a(\zeta) + hb(\zeta))\hat{y}^n = \bar{a}(\zeta)\hat{y}^{n-1}, \qquad (3.4)$$

whose transfer function

$$K_h(\zeta) = \frac{\bar{a}(\zeta)}{a(\zeta) + hb(\zeta)} \tag{3.5}$$

gives the essential information about the rate of convergence for (3.1), too.

As shown in [7] the convergence rate is given by

$$\sup_{|\zeta|>1} |K_h(\zeta)|. \tag{3.6}$$

As we know that $\rho(K(i\xi)) \to 1$ for the time continuous iteration when $|\zeta| \to \infty$, we here want to study (3.6) as a function of time step h.

Example 3.1 Let us use backward Euler in (3.1) and choose $\bar{a} = a$. Then the discrete equation, where $y_j \approx y(jh)$, becomes

$$y_j^n = \frac{1}{1+h}(y_{j-1}^n + y_j^{n-1} - y_{j-1}^{n-1}), \ y_0^n = 0 \ \forall n. \ (3.7)$$

For the first step we get iteration

$$y_1^n = \frac{1}{1+h} y_1^{n-1} \, .$$

It converges to zero with the rate $\frac{1}{1+h}$, which approaches 1 as $h \to 1$. Thus mesh refinement would slow down convergence. The maximum in (3.6) for backward Euler can be easily computed as $\max_{\theta} \{a(e^{i\theta})/(a(e^{i\theta}) + hb(e^{i\theta}))\} \approx 1 - h/2$, and is reached at $\theta = \pi$.

Let us now study $K_h(\zeta)$ given in (3.5). If $\bar{a} = a$ we can write (assuming $a(\zeta) \neq 0$)

$$K_h(\zeta) = \frac{1}{1 + hb/a(\zeta)}.$$

One sees immediately that for instance with trapezoidal rule $|K_h(\zeta)|$ reaches the value 1 since $b(e^{i\pi}) = 0$ for this method. As the connection between the Laplace variable $i\xi$ and ζ is $\zeta = e^{i\xi h}$, we may consider the term $hb/a(e^{i\xi h})$ for small values of (ξh) using the usual order conditions for multistep methods, see e.g. [2]

$$\frac{1}{h}\frac{a}{b}(e^{i\xi h}) = i\xi(1+c_p(i\xi h)^p + c_{p+1}(i\xi h)^{p+1} + \ldots)$$
(3.8)

where p is the order of the method (a, b) and c_p the error constant. Inverting (3.8) we get

$$h\frac{b}{a}(e^{i\xi h}) = \frac{1}{i\xi}(1+\delta_1(i\xi h)^p + \delta_2(i\xi h)^{p+1} + \dots). (3.9)$$

If $\bar{a} \neq a$ is used then it may be possible to improve the rate of convergence since $K_h(\zeta)$ now becomes

$$K_h(\zeta) = \frac{\bar{a}/a(\zeta)}{1 + hb/a(\zeta)}.$$
 (3.10)

If such \bar{a} can be chosen that $|\frac{\bar{a}}{a}(e^{i\xi h})|$ is smaller than 1 then the absolute value of K_h can be diminished. In addition, \bar{a} must be such that the order of the multistep method is preserved, i.e. for small values of (ξh)

$$\frac{\bar{a}}{a}(e^{i\xi h}) = 1 + \gamma_1(i\xi h)^p + \gamma_2(i\xi h)^{p+1} + \dots \quad (3.11)$$

Let us now approximate (3.10) for first order methods.

First order methods p = 1

The real and imaginary parts of (3.9) and (3.11) become

$$\frac{\bar{a}}{a}(e^{i\xi h}) = (1 - \gamma_2(\xi h)^2 + \cdots) + i(\gamma_1(\xi h) + \cdots)$$

$$h\frac{a}{b}(e^{i\xi h}) = (\delta_1 h + \cdots) + i(-\frac{1}{\xi} + \delta_2 \xi h^2 + \cdots).$$

Notice that for A-stable methods δ_1 must be nonnegative. So letting c denote a positive constant we can approximate

$$|K_h(e^{i\xi h})|^2 = \frac{1 + (\gamma_1^2 - 2\gamma_2)(\xi h)^2 + \mathcal{O}((\xi h)^4)}{1 + 2\delta_1 h + \frac{1}{\xi^2} + \mathcal{O}(h^2)} \le \frac{1}{1 + ch},$$

where the last inequality holds if $\delta_1 > 0$, $\gamma_1^2 - 2\gamma_2 \le 0$ and $|\xi| \le \frac{1}{\sqrt{h}}$. What happens for large values of ξ depends then on stability region and \bar{a} . For example for backward Euler (with $\bar{a} = a$) we have

$$|K_h(e^{i\xi h})| \leq \frac{1}{1+h/2}$$
 for all ξ .

If one combines the filter $\bar{a} = \frac{1}{2}(\zeta - \zeta^{-1})$ with backward Euler then the constant multiplying h can be improved. The discrete iteration (3.7) is modified to

$$y_j^n = \frac{1}{1+h}(y_{j-1}^n + (y_j^{n-1} - y_{j-2}^{n-1})/2), \ y_0^n = 0 \ \forall n.$$

Maximization in (3.6) gives now

$$\max_{\theta} \left| \frac{\left(e^{i\theta} - e^{-i\theta} \right)/2}{e^{i\theta} - 1 + he^{i\theta}} \right| = \frac{1}{1+h},$$

and the maximum is obtained at a small value of θ with $\cos \theta = \frac{1}{1+h}$. The constants in (3.11) satisfy $\gamma_1^2 - 2\gamma_2 =$

 $-\frac{1}{4}$ and $\delta_1 > 0$ since the method is A-stable. We get gain in speed from 1 - h/2 to 1 - h but we lose in the error constant respectively.

Second order methods p = 2

Here (3.11) and (3.9) become

$$\frac{\bar{a}}{a}(e^{i\xi h}) = 1 - \gamma_2(\xi h)^2 + \dots + i(-\gamma_3(\xi h)^3 + \dots)$$

$$h\frac{a}{b}(e^{i\xi h}) = -\delta_3\xi^2 h^3 + \dots + i(-\frac{1}{\xi} + \delta_2\xi h^2 + \dots).$$

where $-\delta_3$ is nonnegative for A-stable methods. So we have

$$|K_h(e^{i\xi h})|^2 = \frac{1 - 2\gamma_2(\xi h)^2 + \mathcal{O}((\xi h)^4)}{1 - 2\delta_3\xi^2h^3 + \frac{1}{\xi^2} + \mathcal{O}(h^2)}$$

and $|K_h|$ grows from 0 to 1/(1+ch) as ξ grows from 0 to $1/\sqrt{h}$ provided $\gamma_2 > 0$. Again what happens for large values of ξ depends on stability region and \bar{a} . We give some examples which show that it is possible to choose \bar{a} in such a way that $|K_h| \le 1/(1+ch)$.

Example 3.2 Let us use BDF2 method and 4-step BDF2 as the filter: $\bar{a}(\zeta) = \frac{1}{2}a(\zeta^2)/\zeta^2$, where $a(\zeta) = \frac{3}{2}\zeta^2 - 2\zeta + \frac{1}{2}$ and $b(\zeta) = \zeta^2$. The discrete iteration is now

$$y_j^n = \frac{1}{3/2 + h} (2y_{j-1}^n - y_{j-2}^n/2 + 3y_j^{n-1}/4 - y_{j-2}^{n-1} + y_{j-4}^{n-1}/4).$$

It is straightforward to compute that

$$\left|\frac{\bar{a}}{a}(e^{i\theta})\right|^2 = 1 - \frac{\sin^2\theta}{1 + 4\sin^2\theta}$$

which implies that $\left|\frac{b}{a}(e^{i\xi h})\right| \leq 1$ always and especially for small (ξh)

$$\left|\frac{\bar{a}}{a}(e^{i\theta})\right| \approx 1 - (\xi h)^2$$
.

We may also compute that

$$\max_{\theta} \left| \frac{1}{1 + hb/a(e^{i\theta})} \right| \leq \frac{1}{1 + ch}$$

so that we may conclude that

$$|K_h(e^{i\theta})| \leq \frac{1}{1+ch}$$

holds for all θ . As without the filter we have at $\frac{1}{\ell^2} = h^{3/2}$

$$\left|\frac{1}{1+hb/a(e^{i\xi h})}\right| \sim \frac{1}{1+ch^{3/2}}$$

the acceleration is now of qualitative nature.

Example 3.3 A natural choice for a filter to be used with the trapezoidal rule would be derivative approximation over 3 time steps symmetrically with respect to the point h(j-1/2):

$$\bar{a}(\zeta) = \frac{1}{3}a(\zeta^3)/\zeta = \frac{1}{3}(\zeta^2 - \zeta^{-1})$$

where $a(\zeta) = \zeta - 1$ and $b(\zeta) = \frac{1}{2}(\zeta + 1)$ for the trapezoidal rule. Here the discrete equation would be

$$y_j^n = \frac{1}{1 + h/2} ((1 - h/2) y_{j-1}^n + (y_{j+1}^{n-1} - y_{j-2}^{n-1})/3).$$

One can compute the approximation

$$\frac{\bar{a}}{a}(e^{i\theta}) = 1 - 6\theta^2 + \mathcal{O}(\theta^3)$$

for small θ so that this filter would seem to improve conergence for small (ξh) . Also

$$\left|\frac{\bar{a}}{a}(e^{i\theta})\right| = \frac{1}{3}|e^{i2\theta} + e^{i\theta} + 1| \le 1.$$

However, since $K_h(\zeta)$ is here

$$K_h(\zeta) = \frac{(\zeta^2 - \zeta^{-1})/3}{\zeta - 1 + h(\zeta + 1)/2},$$

we notice that $K_h(\zeta)$ is not bounded as $\zeta \to \infty$. The reason is that the filter \bar{a} is here such that when computing the value $y_j^n \approx y^n(hj)$ we use the value y_{j+1}^{n-1} from the previous iteration. In practice this leads to difficulties in determining the initial values for the sweeps.

One may, of course, consider other methods and construct filters to them, e.g. Example 3.2. suggests that the filter $\bar{a}(\zeta) = \frac{1}{2}a(\zeta^2)/\zeta^k$ would work for BDF methods, but we have here considered only the A-stable methods which have order p < 2.

4. ACCELERATION

As the convergence is quite slow, acceleration is important. The following possibilities, at least in in principle, can be used.

- (i) preprocess the inital guess
- (ii) regularize the problem
- (iii) play a game with gradual mesh refinement.

- (i) It was demonstrated at the end of Section 2 that if one can prepare the initial guess in such a way that its derivatives are correct up to order l-1, then we can expect the convergence of the form $\mathcal{O}(\frac{1}{n^r})$ with $r \sim l/2 + 1/4$. So all one has to do is to "solve" the problem over a tiny interval and then extend this solution smoothly for larger time values.
- (ii) There are several ways to regularize a "nearly singular" problem. For the present example a natural trick would be to include an extra capacitor C_1 , see Figure 2, in the early iterations and let gradually $C_1 \rightarrow 0$.
- (iii) A "tolerance game" has been analyzed in [9] for "short window superlinear convergence" and in [8] for "long window geometric convergence". The basic idea is to balance the discretization error and the iteration error while computing. Here this is rather easy as the convergence is slow (i.e. $\rho \sim 1-h$) and as we proceed, the step size reductions occur less and less frequently. When $\rho \approx \text{const} < 1$, the step size reductions take place repeatedly and the problem arises whether one can reliably make decisions on step size reductions without effecting the actual rate of convergence. This has been analyzed in [8] in detail.

The actual problem shows up in the interpolation of coarse mesh couplings. It has been shown that there are stable and reliable ways to interpolate at any order. Here we can omit this problem.

Let us first estimate the amount of work needed to solve a model problem with fixed time step h, when the order of the method is p and we assume the model $\rho \sim 1-h$. Let ϵ denote the tolerance we are interested in. Then one sweep takes $h^{-1} \sim (\frac{1}{\epsilon})^{1/p}$ time points. Now, as $\rho \sim 1-h$, we have with h^{-1} sweeps the error down in $\frac{1}{\epsilon}$. Thus

$$\nu \approx \frac{\log \frac{1}{\epsilon}}{h} \approx \log \frac{1}{\epsilon} (\frac{1}{\epsilon})^{1/p}$$

sweeps will take the initial error of order 1 down to level ϵ .

Total amount of work W_0 is proportional to ν/h and thus:

$$W_0 \sim (\frac{1}{\epsilon})^{2/p} \log \frac{1}{\epsilon}.$$

Let us compute a similar estimate with the mesh refinement. Assume that (e.g.) we use the scale $h_j = e^{-j}$. As the method is of the order p the discretization error is proportional to h_j^p and thus with step size h_j we iterate so long that the iteration error gets reduced by the factor e^{-p} . As, by then, $\rho \sim 1 - e^{-j}$, this means $\sim e^j$ sweeps and the total work with step size h_j is proportional to pe^{2j} . We get down to tolerance level when $h_N^p \sim \epsilon$, i.e.

for $N \sim \frac{1}{p} \log \frac{1}{\epsilon}$. Thus the total amount of work, say W_2 , satisfies

$$W_2 \sim pe^{2N} \{1 + e^{-2} + e^{-4} + \cdots \}$$

 $\sim \frac{p}{1 - e^{-2}} (\frac{1}{\epsilon})^{2/p}$

which means that the gain is a factor $\log \frac{1}{\epsilon} / \frac{p}{1 - \epsilon^{-2}}$ over W_0 . This is the same gain as what one obtains in the geometrically converging case [8].

REFERENCES

- P. Defebve, F. Odeh, A. E. Ruehli, Waveform Techniques, in Circuit Analysis, Simulation and Design, Part 2, edited by A. E. Ruehli, Elsevier, North-Holland 1987.
- [2] E. Hairer, S. P. Nørsett, G. Wanner, Solving Ordinary Differential Equations I, Springer 1987.
- [3] B. Leimkuhler, U. Miekkala, O. Nevanlinna, Waveform Relaxation for Linear RC-Circuits, Helsinki University of Technology, Institute of Mathematics, Research Reports A272, November 1989.
- [4] E. Lelarasmee, A.E. Ruchli, A.L. Sangiovanni-Vincentelli The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits, IEEE Trans. Computer-Aided Design of ICAS, vol. CAD-1, no.3, pp.131-145, 1982.
- [5] U. Miekkala, O. Nevanlinna, Convergence of Dynamic Iteration Methods for Initial Value Problems, SIAM J. Sci. Stat. Comp., Vol. 8, No. 4, 1987.
- [6] U. Miekkala, O. Nevanlinna, Convergence of Waveform Relaxation Method, Proceedings of 1988 IEEE International Symposium on Circuits and Systems, Helsinki, June 7-9, 1988, pp. 1643-1646.
- [7] U. Miekkala, O. Nevanlinna, Sets of Convergence and Stability Regions, BIT 27 (1987), 554-584.
- [8] O. Nevanlinna, Power Bounded Prolongations and Picard-Lindelöf Iteration, Helsinki University of Technology, Institute of Mathematics, Research Reports A271, October 1989.
- [9] O. Nevanlinna, Remarks on Picard-Lindelöf iteration, part I, BIT 29 (1989), pp. 328-346, part II, BIT 29 (1989), pp. 535-562.
- [10] J. White, A. Sangiovanni-Vincentelli, Relaxation Techniques for the Simulation of VLSI Circuits, Kluwer Academic Publishers, 1987.

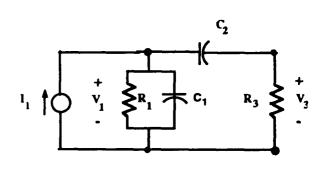


Figure 2.

Partitioning Tradeoffs for Waveform Relaxation in Transient Analysis Circuit Simulation

Lena Peterson
Sven Mattisson
Department of Applied Electronics
Lund University PO Box 118
S-22100 Lund, SWEDEN
email: lena@tde.lth.se sven@tde.lth.se

Abstract

In this paper we present a study of static circuit partitioning algorithms for waveform relaxation used for circuit simulation on a multicomputer. We investigate the important tradeoff between the irregularity of the partitioning and the achievable parallelism. Also, the importance of the accuracy in certain steps in the partitioning calculation has been studied. Purely topological methods are compared with methods that try to quantitatively estimate the convergence factor of the WR iterations. For digital CMOS circuits we find that only the close neighborhood of a circuit node (nearest neighbors) influences the value of the worst-case coupling. The conductive coupling is the essential part to take into account for the partitioning. For a few circuits (mostly circuits with cross-coupling) the capacitive coupling must also be included in the partitioning. It is, however, hard to find an exact limit for the convergence factor estimate.

1 Introduction

It is extremely CPU-time consuming to perform accurate timing verification for the large integrated circuits that are possible to fabricate today. For many circuits the simulation run times have been reduced by the use of logic simulators and switch-level simulators. However, circuit designers still need circuit simulation programs that do not trade their accuracy for speed, both to simulate digital and analog circuits. One way to shortening the circuit simulation run times, without sacrificing the accuracy, is to use concurrent computers. The traditional algorithms used for circuit simulation a include the solving of a large linear system. This part of the simulation program accounts for 10 – 25 % of the total runtime, depending on the circuit size,

and is not easily parallelizable. Thus, when we want to use more than 4 - 10 processors efficiently (according to Amdahl's law) we have to turn to alternative algorithms.

One highly promising algorithm, which is used for circuit simulation and also for other applications, is the waveform relaxation (WR) method [4, 2]. This algorithm requires that the equation system to be solved is partitioned into subsystems. For circuit simulation, to partition the equation system is really the same as to partition the circuit to be simulated. The objective of this study is to investigate several partitioning algorithms and specially their impact on run times and achievable parallelism for WR run on a multicomputer. By multicomputer we mean a concurrent message-passing computer that has a local memory space for each processing node.

2 Waveform Relaxation

Waveform relaxation is an iterative method that can be used for performing transient analysis. That is, it is a method that can solve the equations formed by applying Kirchoffs current law (or Kirchoffs voltage low, or both) to the description of a circuit. These equations, representing the circuit dynamics, form a system of ordinary differential equations (ODEs). Such a system can be partitioned into several subsystems each containing at least one ODE. Using the waveform relaxation method, one can solve these subsystems with only little interaction between the subsystems. When solving one of the subsystems all the other ones are relaxed, that is, the solutions from these subsystems are assumed to be static. This way, it is possible to solve for the state variables of one subsystem over the total simulation interval without interacting with the other subsystems. Thus, one

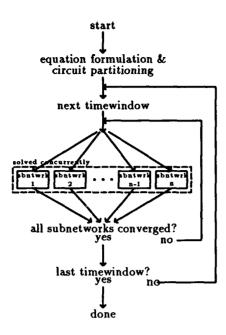


Figure 1: Program flow for the waveform relaxation method.

obtains functional approximations of all the state variables over the simulation interval, that is, waveforms for the state variables. Each global WR iteration consists of one such solution round for all the subsystems. Different iteration schemes may be used for the WR iterations, the most common ones are Gauss-Seidel and Jacobi iterations. The partitioning of the ODE system is extremely important for the convergence speed of the WR method and is thus important for the successful use of the method.

In a practical implementation of the WR method one does not solve for the total simulation interval in one iteration. Such an approach is extremely memory consuming. Furthermore, much effort is spent on computing the end of the waveforms which do not contain any information during the first iterations. Instead, the total simulation interval is divided into shorter time windows. Each such time window is then solved using the WR method. In our program, CONCISE, the time windows are precomputed from the input waveforms. This approach helps the integration algorithm to home in on discontinuities in the state variable waveforms [5]. The program flow for the WR method is shown in Figure 1.

3 Coupling in an MOS Circuit

The objective of the partitioning methods used for the WR relaxation method is to cluster equations that are strongly coupled into the same subsystem. Thus,

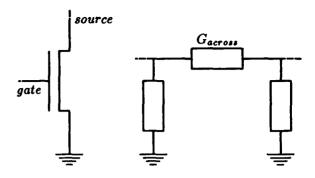


Figure 2: Electrical examples of coupling. The MOS transistor on the left exhibits unidirectional coupling from gate to source whereas the conductor on the right is an example of bidirectional coupling.

the couplings between the resulting subsystems will be loose. The couplings between neighbor circuit nodes in the circuits are of two types, unidirectional and bidirectional couplings. In MOS circuits, the unidirectional couplings are the ones from gate to drain and gate to source. The main bidirectional couplings are the ones between drain and source. Furthermore, there are capacitive bidirectional couplings between the terminals of the MOS transistor. Parasitic capacitors and conductors, which are due to the the internal structures in the integrated circuits, also cause bidirectional coupling.

Widening our view to the entire circuit, we find that there is yet another way to characterize coupling — the coupling may be either local or global. Local coupling is bidirectional coupling between neighbor circuit nodes. The local coupling usually stems from the bidirectional coupling as described above, but it could also be due to two unidirectional couplings, one in each direction. In an MOS circuit the latter type would correspond to a pair of cross-coupled transistors. Global coupling is due to a closed loop of unidirectional couplings. There is also the less severe case of an non-closed loop of unidirectional couplings.

In this study we have concentrated on the local coupling stemming from bidirectional couplings between circuit nodes. The local coupling due to unidirectional coupling is easily taken care of by a routine that scans the circuit for cross-coupled transistors.

The global coupling need an entirely different approach from the local. Usually the closed loops are found by dataflow or graph algorithms [2]. Such a method is not yet included in our program CONCISE. The nonclosed chains of unidirectional couplings are usually taken into

account when iteration schemes of Gauss-Seidel type are used. The equations are then ordered according to the directions of these unidirectional couplings. For the experiments described in this paper we have used Jacobi iterations and thus we have not considered this type of coupling.

4 Partitioning Methods

The partitioning methods used for circuit simulation applications can be divided into two groups. The first group contains purely topological methods, that is methods that only take the structure of the circuit into account when partitioning the circuit. In addition to the structure of the circuit, the methods in the second group also consider quantitative information, for example the values and sizes of components in the circuit, when partitioning the circuit.

Topological Methods

Most of the methods in the first group are rather similar. The idea behind these methods is to cluster circuit nodes which at DC have a conducting path between them. For purely digital circuits without parasitic conductances this partitioning strategy is equivalent to clustering circuit nodes connected by the source and drain of an MOS transistor. Thus, in this paper we call our version of this method the source-drain partitioning method. Methods of this DC-path type for the WR algorithm have been described in [2, 3] and the circuit partitioning method used in the switch-level simulator MOSSIM [1] is also similar. These methods are motivated by the unidirectionality of the MOS transistor and are therefore mainly suited for MOS circuits.

Quantitative Methods

The methods of the second group try to estimate how fast the WR iterations will converge. To explain how to arrive at such an estimate we must take a closer look at the WR method. The convergence proofs for the WR method prove the method to be a contraction map in waveform space under certain fairly easily-full-filled conditions. That is, WR is shown to be a map F in a waveform space Y so that $F(Y) \in Y$ and for some norm

$$|| F(y) - F(x) || \le \gamma || x - y ||$$
 for all $x, y \in Y$
 $\gamma \in [0, 1]$

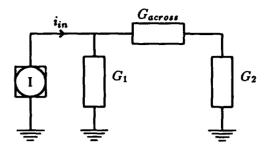


Figure 3: Simple 2-node circuit that exhibits bidirectional coupling.

holds. For such a contraction map the rate of convergence is defined as

$$||y^k - y^*|| \le \gamma^k ||y^\circ - y^*||,$$

where y° is the initial waveform and y^{*} is the fixed point of the iteration (that is the solution). Consequently, γ is called the convergence factor.

For the methods in the second group the goal is to compute an estimate of the resulting convergence factor for each pair of circuit nodes if these nodes were limiting for the convergence factor for the total circuit. The two circuit nodes are clustered if the estimate is higher than a user-specified threshold value. Usually, the estimate of the convergence factor is computed for the worst case circuit state since it impossible to know what the the internal state of the circuit will be during the simulation. Thus, these methods are often overly pessimistic. A typical-case estimate would probably be a more appropriate for concurrent implementation of WR where the price for large subsystems is high. However, such an estimate is hard to calculate since it is impossible to know what the typical state for the circuit is. Thus, we have remained at the worst-case estimate in this investigation.

5 Calculation of the Convergence Factor Estimate

Each pair of circuit nodes that exhibit bidirectional coupling may be viewed as a 2×2 -system. The coupling of the circuit nodes is modeled as a nonlinear admittance connecting the circuit nodes. The impact of the rest of the circuit at each of the circuit nodes is lumped in the Norton equivalent admittance, that is the same as the driving point admittance.

For linear systems the convergence constant is the spectral radius of the iteration matrix. For both the Gauss-

Seidel and the Jacobi iteration schemes this expression is of the same form, only differing in the square root present in the Jacobi case. Thus, we get the following expression as an estimate for the coupling

$$\gamma_{coup} = \frac{|Y_{across}|}{|Y_1 + Y_{across}|} \cdot \frac{|Y_{across}|}{|Y_2 + Y_{across}|}.$$
 (1)

By using the largest possible value for the admittance between the nodes and the smallest for the two driving point admittances one obtains a worst-case value for the convergence factor.

6 The Diagonal Dominance Norton Method

The above method of estimating the convergence factor was first described as the diagonal dominance Norton (DDN) partitioning in [11]. When one uses a difference approximation for the derivative, the approximation used in the Jacobian matrix for a linear admittance is

$$G + sC \approx G + \frac{a_0}{h}C,$$
 (2)

where $\frac{a_0}{h}$ is the derivative operator consisting of h, the current time step and a_0 the zeroth coefficient of the integration formula. At DC (h large), the conductance part is dominating but at transients (h small) the capacitance will dominate the coupling. Thus, in [11] the coupling is estimated for the two extremes, firstly when the conductance is totally dominant (frequency is 0), and secondly when the capacitances are totally dominant (frequency is high). This way, the capacitive coupling and the capacitive coupling are computed separately as

$$\gamma_{cond} = \frac{G_{across}}{G_1 + G_{across}} \cdot \frac{G_{across}}{G_2 + G_{across}}, \qquad (3)$$

and

$$\gamma_{cap} = \frac{C_{across}}{C_1 + C_{across}} \cdot \frac{C_{across}}{C_2 + C_{across}}.$$
 (4)

The conductive and capacitive clusterings are performed separately. The compound partitioning is the union of the both partitionings such that two nodes that belong to the same subnetwork in either of the two partitionings has to belong to the same subnetwork in the compound partitioning.

As mentioned above, the internode admittance should be maximized and the node-to-ground admittance minimized to find the worst-case coupling. Let us consider the pure conductance coupling as in equation (3). The minimum conductance between source and drain of an MOS transistor is identical to zero. Thus, for circuits where the only conductive elements are source-drain connections, the coupling when conductance is dominating (at DC) will be identical to one. Subsequently, the conductance part of the DDN algorithm is identical to the topological S-D method for circuits where the only conductances are those of the MOS transistors. Thus, the DDN method adds the effect of the capacitive coupling to the S-D method for such circuits.

7 The Admittance Matrix Method

As noted in [11], the DDN algorithm is pessimistic and may give unnecessarily large subnetworks. We have already discussed the problem of using the worst-case convergence factor. One way of finding a more realistic value for this worst-case coupling is to use the smallest and largest time step permitted by the simulation program for the two extremes. This change has been suggested in [2]. Another approach is to use the intrinsic time constant of the components to find the highest possible time constant of the signals inside the circuit. We have chosen the latter approach for our admittance matrix method.

8 Computing the Driving Point Admittance

The computationally most expensive part of all methods that use the convergence factor estimate from equation 1 is to calculate the driving point admittance values, that is G_1 and G_2 in Figure 1. In [11] a recursive depth-first algorithm for this calculation is given. For a ladder-type circuit this algorithm computes the correct admittance, but for other topologies it only gives an approximation since it includes each node only once. The authors note in [11] that the recursion will not be deep since the minimum conductance of the source drain connection is zero. This observation holds for the purely conductive case, but when capacitances are included the recursion will continue throughout the circuit.

For the admittance method (AM) we have implemented an algorithm that gives the correct driving admittance for a node in an arbitrary linear circuit. The circuit considered is the original one with the across-admittance removed and all other connections replaced by their minimum admittance. When Y denotes the admittance matrix formulated by nodal analysis the driving point

admittance can be computed as

$$\frac{\det(Y)}{\det(Y_j^j)} \tag{5}$$

where Y_j^j is the first-order cofactor of Y with respect to row and column j [6].

In our implementation the matrix size can be limited by only including circuit nodes a limited number of neighbor hops from the original circuit node. We denote the resulting methods AMO (0 hops), AM1 (1 hop), and so on to AM ∞ (the entire circuit). The matrix method is computationally highly expensive when the matrix is large but by using this method we can get an impression of how important distant circuit nodes are to the resulting admittance value.

One interesting observation is that by taking more nodes into account (deeper recursion or larger matrix) when computing the driving point admittance this admittance can only increase, not decrease. That is, the coupling will always decrease when we look further into the circuit. Thus, we can start our coupling calculation by computing an approximate value for the coupling. This we do by taking only the admittances connected directly to the node in question into account. If this approximate coupling is lower than the threshold for clustering a thorough calculation of the driving point admittance is unnecessary. This way the computational requirements can be lowered both for the matrix method and the depth-first algorithm.

9 The Program

The program used for the test, called CONCISE, is a circuit simulator for transient analysis of CMOS circuits. It is written in C and uses the Cosmic Environment/Reactive Kernel message-passing primitives [10]. These primitives support the programming model where each process has its own memory-space. This model makes dynamic partitioning and load balancing CPU-time expensive and thus the study was limited to static schemes where the partitioning and placement remain fixed throughout the computation. It is important to notice that the requirements on the partitioning algorithms in this case differ from the "traditional" parallelization where only a few processing nodes are used. The load balancing considerations become much more difficult when the number of processing nodes are about the same as the number of circuit nodes. The computer used in the study was a 64-node Symult s2010.

Table 1: Description of the circuits used for the experiments.

Circuits used for the experiments							
circuit	trans	nodes	description				
adder	262	105	4-bit-wide slice of an				
~~~~	202	100	NMOS multiplier using				
			the Booth algorithm.				
contr	701	355	Control part of signal				
CORUL	,01	000	processing chip. This cir-				
			cuit is comprised of the				
[			circuits proc, inblock				
[			and sign.				
dram	793	535	Dynamic RAM with 7-bit				
		000	address and 3-bit data.				
delay	2844	1944	4 shift register delay lines				
,		2022	of 128 stages each and 16				
]			8-to-1 demultiplexers and				
ļ			1 2-to-4 demultiplexer.				
inblock	221	119	Mixed parts for the con-				
			troller. A small PLA (3				
			inputs, 2 outputs), 2 full-				
			adders, 5 latches, and 12				
			AND gates.				
mult	3134	1739	8 × 8 multiplier using the				
			Booth algorithm.				
pla	1428	170	Pseudo-NMOS PLA with				
			5 inputs, 64 outputs, and				
			32 rows.				
proc	240	92	Finite state machine us-				
ł			ing pseudo-NMOS PLA				
			with 10 inputs, 6 outputs,				
			and 23 rows. 5 latches for				
Teg	1920	1152	the state. Two 64-bit wide shift reg-				
reg	1920	1102	isters with 64 latches be-				
			tween.				
regpla	3348	1322	The circuits reg and pla				
- 01-4	0010	1022	combined.				
ram	209	122	4-bit NMOS RAM. De-				
			signed with hot-clock				
ļ	!		techniques.				
ram2	1153	625	7-bit RAM with 7-bit ad-				
			dress and 3-bit data.				
rom	414	240	ROM with 7-bit address				
			and 3-bit data.				
sign	240	144	Two 8-bit wide shift reg-				
]			isters (with parallel load)				
			with 8 latches between				
			them.				

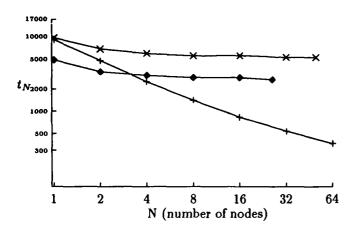


Figure 4: Runs for adder circuit. The partitionings are pointwise partitioning (+), source-drain partitioning (×), and DDN partitioning depth 0 with  $\gamma = 0.1$  (•).

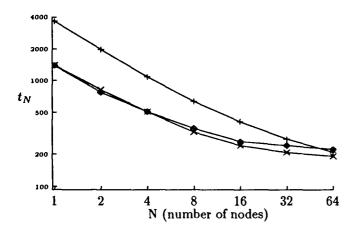


Figure 5: Runs for inblock circuit. The partitionings are pointwise partitioning (+), source-drain partitioning (×), and DDN partitioning depth 0 with  $\gamma = 0.02$  ( $\spadesuit$ ).

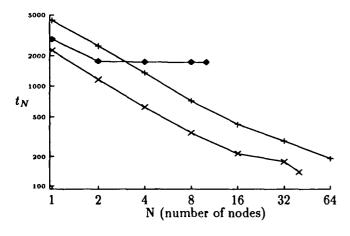


Figure 6: Runs for sign circuit. The partitionings are pointwise partitioning (+), source-drain partitioning (×), and DDN partitioning depth 0 with  $\gamma = 0.01$  ( $\spadesuit$ ).

Table 2: The convergence factor threshold at which the capacitive partitioning starts merging nodes for various methods of computing the driving point admittance. DDN is the method proposed in [11] and AM is the admittance matrix method.

Val	Value for $\gamma_{min}$ for capacitive partitioning								
	ddn0	$\mathrm{ddn}\infty$	am0	am l	am2				
adder	0.13	0.12	0.13	0.11	0.11				
contr	0.02	0.02	0.02	0.02	0.02				
dram	0.08	0.07	0.08	0.08	0.08				
delay	0.04	0.04	0.04	0.03	0.03				
inblock	0.02	0.02	0.02	0.02	0.02				
mult	0.10	0.09	0.10	0.10	0.10				
pla	0.001	0.001	0.001	0.001	0.001				
proc	0.009	0.009	0.009	0.009	0.009				
reg	0.01	0.01	0.01	0.01	0.01				
regpla	0.01	0.01	0.01	0.01	0.01				
ram	0.09	0.09	0.09	0.09	0.09				
ram2	0.07	0.07	0.07	0.07	0.07				
rom	0.08	0.07	0.08	0.08	0.08				
sign	0.01	0.01	0.01	0.01	0.01				

## 10 Experiments

The circuits used in the experiments are all digital MOS circuits. All in all there are 14 circuits. Two of them, adder and ram, are 4µ NMOS circuits that employ hotclock techniques [9]. The other twelve test examples are 2u CMOS circuits. All fourteen of them come from research chips designed either in Lund or at Caltech. The circuits are described in table 1. The netlists for all the circuits were extracted from the chip layouts by the built-in extractor in the chip-design system Magic [8]. This extractor extracts internode capacitances, but not internode resistances. Furthermore, the extractor does not extract the size of the source and drain areas of the MOS transistors. These areas are needed since they define the sizes of the source-bulk and drain-bulk diodes. For our test circuits we have used the same size for all these diodes.

All the circuits have been simulated using CONCISE with pointwise partitioning (one ODE per subnetwork), source-drain partitioning, and with the different methods that also consider the capacitive coupling.

## Capacitive Coupling

We would like to investigate the difference between the coupling values computed by the various methods which consider the capacitive coupling. To get a picture of the

Table 3: Comparison between pointwise, source-drain(conductive), and DDN(conductive and capacitive) partitioning. The CPU time is the for the transient analysis part run in 1 node except where stated otherwise.

<u> </u>	p	ointwise			source-	drain				DDN0	····	
circuit	CPU	iter/	ckt	CPU	iter/	# of	max	$\gamma$	CPU	iter/	# of	max
	time	wind	nds	time	wind	subs	size		time	wind	subs	size
adder	8987	19.41	105	9550	9.55	50	20	0.07	4890	4.20	26	24
contr	2294	13.12	355	6744	4.07	175	7	0.01	8929	3.57	141	11
delay ¹	39636	15.43	1944	16886	3.07	632	9	0.04	14908	2.71	600	21
dram	5838	10.07	535	15223	5.74	118	50	0.07	13211	6.00	116	50
inblock	3659	9.69	119	1399	3.60	67	7	0.02	1389	3.51	66	8
mult ²	138852	18.06	1739	65998	7.12	700	11	0.04	75851	6.54	537	13
pla	2622	4.00	170	2622	4.00	170	1	0.001	5272	4.00	115	2
proc	3949	10.31	92	1888	3.75	68	3	0.009	2649	3.38	62	6
ram	14539	22.79	122	2255	4.07	64	7	0.09	2310	4.04	62	7
ram24	32654	19.47	625	41311	8.50	119	72	0.05	36969	8.00	117	72
reg ³	23512	11.80	1152	10438	2.86	325	5	0.01	12432	2.52	197	8
regpla ¹	72590	19.64	1322	23019	4.58	428	5	0.01	44949	5.30	300	8
rom	12400	19.45	240	4488	8.46	103	8	0.08	4245	7.92	102	8
sign	4431	11.88	144	2263	3.15	40	5	0.01	2896	2.18	10	64

- 1. CPU time is total for run in 8 nodes.
- 2. CPU time is total for run in 16 nodes.
- 3. CPU time is total for run in 4 nodes.
- 4. CPU time is total for run in 2 nodes.

size of the capacitive couplings we decreased the convergence factor threshold in small steps (usually 0.01) until nodes were merged due to the capacitive coupling for these various methods. The resulting convergence factor thresholds are shown in Table 2. Comparing the resulting partitionings from the methods that consider the capacitive couping, we find only small differencies when we use the same  $\gamma$ . The major differencies that can be found when different capacitive methods are used is due to the  $\gamma$  threshold used. Only minor differencies are found due to the partitioning method used when the same threshold is used. Thus, we will only consider one method when investigating the capacitive coupling methods compared to the source-drain and pointwise partitioning methods.

# Computing the Driving Point Admittance

From Table 2 is seems probable that the impact of faraway circuit nodes on the calculated coupling is small. To further investigate the the impact of such distant circuit nodes on the calculated coupling we compared the exact values for the driving point admittance with approximate values. This we did for the circuits which we had found to have severe capacitive coupling and for the circuit nodes in these circuits where merging due to capacitive coupling occurred. The exact admittance values where calculated using the admittance matrix method with infinite matrix size (that is including all of the circuit), this we call AM $\infty$ . The approximate values where computed using the AM method but with limited matrix size (AM0, AM1, and AM2). In all cases we came within 0.1depth 1 was used. This means one needs only include the circuit node itself and its nearest neighbors in the driving point admittance calculation.

## Run Times and Convergence Rate

We would like to experimentally verify the positive effects on the convergence rates of the more sophisticated partitioning methods. There is no way to directly measure the convergence rate. However, it is still possible to get an approximate value of the convergence rate from the mean number of WR iterations needed per window which is a fairly good estimate of the convergence rate. From Table 3 we find that the S-D partitioning increases the convergence speed with a factor 2-5.5 over pointwise

^{*}When convergence is slow time windows are split to try to improve convergence. Thus, the convergence rate estimate is too low in these cases. This incorrectness will make the improvements due to better partitioning methods look smaller than they really are.

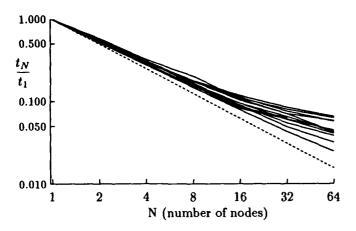


Figure 7: Parallelism, that is inverted speedup for all the circuits for pointwise partitioning. The dotted line is the theoretical limit.

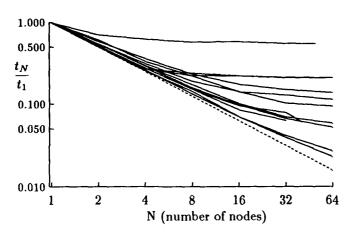


Figure 8: Inverted speedup, for all the circuits for source-drain partitioning. The dotted line is the theoretical limit.

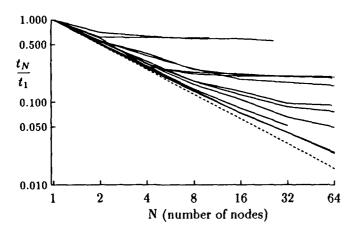


Figure 9: Inverted speedup, for all the circuits for DDN partitioning. The dotted line is the theoretical limit.

partitioning. As expected, an increase in convergence speed gives a decrease in runtime. The run time is, however, not always decreased with as large a factor since the larger subnetworks resulting from source-drain partitioning take longer to evaluate than the trivial ones from the pointwise partitioning. This effect is even more pronounced when the capacitive effect is also taken into account (DDN method). Then the only substantial improvement in execution time is for the adder circuit.

#### **Parallelism**

In Table 3 we find that for some circuits large subnetworks are obtained when S-D or DDN partitioning is used. The achievable parallelism is severely limited for these circuits as can be seen when comparing the diagrams in Figures 7, 8, and 9. In these figures we show curves for all the test circuits together in order to be able to show all the data in limited space. The addition of the part that takes the capacitive coupling into account does increase the convergence speed slightly for most of the circuits but the parallelism is reduced due to large subnetworks in some cases. In Table 3 we find that only for adder does the the inclusion of the capacitive partitioning significantly improve the convergence speed and the execution time. To get a better understanding of the behavior we take a closer look at the curves for three of the circuits in Figures 4, 5, and 6. We also consult Table 3 to see the convergence rate. For adder we find that the S-D partitioning increases the convergence rate but spoils the parallelism. The addition of the capacitive part increases the convergence rate further and reduces the runtime, but of course the parallelism is still poor. It is worth noticing that we had to try several  $\gamma$  thresholds to find the one (0.07) which gives this increase in convergence rate. For inblock the conductive part is the important one. The addition of the capacitive part makes no difference. For sign the addition the capacitive part both increases the runtime and ruins the parallelism. Due to lack of space we have left out the diagrams for the other circuits. These diagrams can be found in [7].

### Speedup

One interesting figure for programs like this one is speedup. The parallelism curves may look discouraging for the more sophisticated partitioning methods. We have compared CONCISE running the traditional direct circuit simulation method, where all the circuit is treated as one large subnetwork, with CONCISE running the WR method. These comparisons may be seen

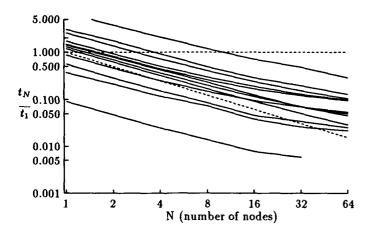


Figure 10: Inverted speedup for CONCISE using the pointwise partitioning normalized to the execution time for CONCISE's direct method (the horizontal dotted line). The sloping dotted line is the ideal parallelization of the direct method.

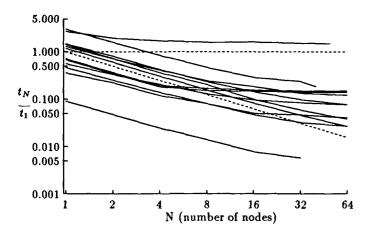


Figure 11: Inverted speedup for CONCISE using the source-drain partitioning normalized to the execution time for CONCISE's direct method (the horizontal dotted line).

in Figures 10 and 11. In these diagrams all the execution times for the WR method are normalized to the execution time of the the direct method, that is "the best sequential algorithm". Thus, we find that the S-D method is not a bad choice even if we could gain even more if the parallelism was not limited by large subnetworks for some of the circuits.

## 11 Conclusions and Discussion

The WR method using pointwise partitioning converges for all our test circuits. However, the convergence is slow for several of them. When the source-drain method or other methods that partition due to conductive coupling is used, the convergence speed is increased for all the test circuits. For a few of the circuits the capacitive coupling is also strong and partitioning that considers capacitive coupling is needed. Thus, we conclude that we need to employ both conductive and capacitive partitioning to get a reasonable convergence speed. For some circuits the partitioning will create large subnetworks, which severely reduce the achievable parallelism. For some circuits dynamic partitioning, which can use information about the current state of the circuit, will help in reducing the size of large subnetworks, but for others it will not. Thus, we need to be able to assign more than one node to perform the calculations for a large subnetwork.

Furthermore, the experiments show the difficulty in finding one fixed convergence constant threshold that is the optimal for all circuits. It is obvious, however, that the admittance calculation need not be extended further than one hop, that is, it should include the circuit node in question and its neighbor circuit nodes. Thus, one can skip the deep recursions in the admittance calculations and use the saved CPU-time for trying several threshold values when considering the capacitive coupling. This can prove useful when running on multicomputers where an jextremely large subnetwork may entirely spoil the achievable parallelism.

# Acknowledgements

We would like to thank Chuck Seitz for his continuous encouragement and support of this work. Thanks also to Wen-King Su for his help in dealing with the Symult s2010. This research is supported in part by the Swedish Board of Technical Development (STU) under grant Dnr 87-3768 and in part by the Defense Advanced Research Projects Agency, ARPA Order number 6202,

and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

## References

- [1] R. E. Bryant, "A switch-level model and simulator for MOS digital systems," Technical report 5065:TR:83, Computer Science Department, California Institute of Technology, Pasadena, 1984.
- [2] P. Debefve, F. Odeh, and A. E. Ruehli, Circuit Analysis, Simulation and Design, vol. 3 part 2 of Advances in CAD for VLSI, ch. 8 Waveform Techniques. Amsterdam, the Netherlands: Elsevier Science Publishers, 1988.
- [3] D. Dumlugöl, The segmented waveform relaxation method for mixed-mode simulation of digital MOS VLSI circuits. PhD thesis, Katholieke Universiteit Leuven, Leuven Belgium, 1986.
- [4] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli, "The waveform relaxation method for time-domain analysis of large scale integrated circuits," IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems, vol. CAD-1, pp. 131-145, July 1982.
- [5] S. Mattisson, CONCISE a Concurrent Circuit Simulation Program. PhD thesis, Department of Applied Electronics, Lund Institute of Technology, Lund, 1986. LUTEDX/TETE-1003/1-116 (1986).
- [6] G. Moschytz, Linear Integrated Networks, fundamentals. New York: Van Nostrand Reinhold, 1975.
- [7] L. Peterson and S. Mattisson, "A Study of Circuit Partitioning Algorithms for Concurrent Waveform Relaxation," Tech. Rep. LUTEDX TETE-7040 1-88 (1989), Department of Applied Electronics, Lund University, Lund, 1989.
- [8] W. S. Scott, R. N. Mayo, G. Hamachi, and J. K. Ousterhout, "1986 VLSI tools: Still more works by the original artists," Tech. Rep. UCB/USD 86/272, Computer Science Division (EECS), University of California, Berkeley, December 1985.
- [9] C. L. Seitz, A. H. Frey, S. Mattisson, S. D. Rabin, D. A. Speck, and J. L. A. van de Snepscheut, "Hotclock nmos," in 1985 Chapel Hill Conference on Very Large Scale Integration, pp. 1-17, 1985.
- [10] C. L. Seitz, J. Seizovic, and W.-K. Su, "The C programmer's abbreviated guide to multicomputer programming," Tech. Rep. CS-TR-88-01, Computer Science Department, California Institute of Technology, 1988. Revised May 1989.

[11] J. White and A. Sangiovanni-Vincentelli, Relaxation Techniques for the Simulation of VLSI Circuits. Boston Dordrecht Lancaster: Kluwer Academic Publishers, 1987.

# Distributed Model Evaluation for the Waveform Relaxation Method

Leif Olsson Lena Peterson Sven Mattisson

Department of Applied Electronics
Lund University PO Box 118
S-22100 Lund, SWEDEN
email: leif@tde.lth.se lena@tde.lth.se sven@tde.lth.se

### **Abstract**

The most time consuming operation in a circuit simulation program is the model evaluation, i.e. the computation of the coefficients for the Jacobian matrix. Since these coefficients only depend on the node voltages and the derivative operator, they can easily be computed in parallel. In this study some methods for computing the Jacobian matrix are discussed. The applicability of the methods with the concurrent waveform relaxation method is also discussed. New results on the complexity of row-wise and column-wise model evaluation, and related stability problems are presented. Experimental results, with respect to efficiency and stability of the coefficient evaluation as well as parallel execution, are given. These experiments have been carried out with the CONCISE circuit simulation program on a Symult s2010 and on a Sequent Symmetry.

### 1 Introduction

A circuit simulation program solves a system of nonlinear ordinary differential equations (ODEs). Each ODE is derived by means of nodal analysis. Thus, the sum of the device currents entering and leaving each circuit node is equated to zero. Each current contribution is computed by evaluating device model equations. By using the node voltages one can compute current contributions from resistors, capacitors, transistors etc.

The system of nonlinear ODEs is traditionally solved by means of a difference approximation of the time derivative, Newton-Raphson iterations for linearization, and LU-factorization for solving the matrix equation in the innermost loop. This scheme is often referred to as a

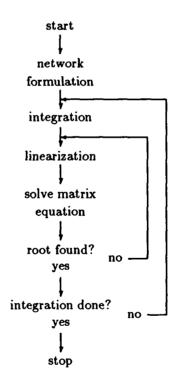


Figure 1: Program flow for transient analysis.

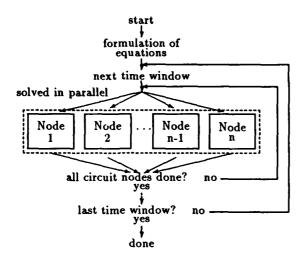


Figure 2: Program flow for the waveform relaxation method.

direct method or a method with global time-step [6], see figure 1.

Concurrent circuit simulation programs often use an iterative method to split the simulation task into subtasks, or subsystems. With such a splitting, all subsystems become decoupled during an iteration and can all be solved in parallel. After each iteration, results are exchanged between subsystems. The iterations continue until consecutive results are sufficiently close, see figure 2.

Compared to the direct method, the iterative method solves the system many times, one for each iteration. However, with the iterative method the matrix inversion becomes very simple and some CPU-time is gained this way. A further enhancement of the iterative method is to use waveform relaxations rather than time-point relaxations [1]. With the former method all iterations are performed on the functional level, that is each ODE subsystem is solved for the entire simulation interval, or a time window, during an iteration. The result of such an iteration is a set of node voltage waveforms. Within each subsystem the integration method can optimize the time steps for its set of ODEs which saves CPU-time. Thus the waveform relaxation method is a multirate integration method.

The combination of multirate integration and simple matrix inversion makes the iterative method comparable in performance to the direct method [4]. In addition it is fairly straightforward to modify the waveform relaxation method to run on a multicomputer [2]. However, special attention has to be paid to the convergence

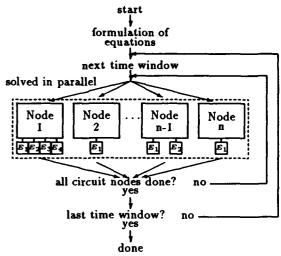


Figure 3: Program flow for the hierarchical waveform relaxation method.

rate of the waveform iterations and the size of the subsystems as slow convergence may lead to excessively long execution times and large subsystems limits the concurrency.

Ideally a subsystem should correspond to one circuit equation, that is a circuit node. In this case, as have been demonstrated with the circuit simulation program CONCISE [2, 3], the available speedup is a fair fraction of the number of circuit nodes. However, if some circuit nodes are strongly coupled, the corresponding equations have to be solved together with a direct method or convergence will be slow [4]. When some smaller subsystems are gathered into one larger subsystem, an equation block, such a subsystem may become much larger than the others. This will result in load imbalance as only one computing node is used per equation block.

It is not always possible to avoid large subsystems. In order to prevent a large subsystem to become a bottle-neck, it is desirable to be able to allocate several computing nodes to the solution of such a large subsystem, see figure 3. This problem is similar to the problem of solving the entire ODE system with a concurrent direct method. The difference is that the largest subsystem typically is much smaller than the entire system if the system itself is large. As have been pointed out earlier [3], there is much less concurrency in a distributed direct method compared to the waveform relaxation method. However, for solving subtasks a distributed direct method may be attractive.

When augmenting the concurrent waveform relaxation method to use two levels of concurrency - parallel evalu-

ation of subsystems and parallel evaluation within subsystems – one must address the partitioning problem as well as decide how much of the direct subsystem solver should be distributed. This study deals with the latter problem, the first problem will be the subject of a future study. Furthermore, as we only consider subsystems of limited size, say below 100 equations, we are primarily interested in an efficient device model evaluation scheme and not a distributed LU-factorization algorithm.

## 2 Computing the Jacobian Matrix

When a subsystem, or block, is evaluated, the computation of the Jacobian matrix coefficients dominates the CPU usage. By parallelizing the device model evaluation, one may obtain a better overall speedup.

The coefficients of the Jacobian are partial derivatives of node currents (function values) with respect to node voltages. These derivatives can be computed by means of a difference approximation, with row-wise or columnwise perturbations, or they can be derived analytically.

When the function values (node currents) stem from a multivalued function, that is, when all function values are computed in a single routine, the column-wise perturbation requires fewer function calls than the rowwise method. The former is  $\mathcal{O}(n)$  and the later is  $\mathcal{O}(n^2)$ . However, in a circuit simulation program, the Jacobian matrix is derived by computing device transfer admittances. Thus, the coefficients are computed by summing contributions from devices. If a single coefficient is to be computed, only the devices connected to the circuit node corresponding to the coefficient in question need to be evaluated. Thus, row-wise perturbation cannot be discarded without further analysis.

#### Definitions and derivations

Let n be the rank of the Jacobian matrix, that is the number of circuit nodes in a particular subsystem. Also, let k be the number of devices connected to circuit nodes in this particular subsystem. Finally, let b be the average number of terminals each device has connected to other circuit nodes inside the subsystem.

From these three basic parameters we can derive two additional interesting characteristics. First we define the density d, that is the average number of devices per circuit node, as

$$d = \frac{kb}{n}.$$
 (1)

Then we define m as the number of nonzero entries in the Jacobian matrix per row. As devices may be connected in parallel it is clear that  $m \leq d$ .

For electrical circuits, each device only has a limited number of terminals. Furthermore, most circuit nodes are only connected with a few neighbor nodes – local interaction. Thus the Jacobian is typically very sparse, and b is in the range of 3-5 for large subsystems. It also follows that k is proportional to n.

#### Row-wise calculation

Let  $C_{row}$  be the number of device model evaluations for the row-wise calculation used in *CONCISE*. Then we need

$$C_{row} = n(1+m)d (2)$$

model evaluations to compute the Jacobian matrix. First  $n \cdot d$  evaluations to calculate the nominal values for all the circuit nodes. Then, to get all the partial derivatives we need to displace each of the other circuit node voltages for each row and recalculate them. Each such calculation costs d and there are m of them per row and we have n rows.

$$C_{row} = n(1+m)d$$

The above expression can be simplified by inserting the definition of d and using the worst-case value for m

$$C_{row} \leq n(1+\frac{kb}{n})\frac{kb}{n} = kb(1+\frac{kb}{n})$$
 (3)

#### Column-wise calculation

With a general column-wise calculation method, the number of device evaluations becomes

$$C_{col} = k(1+n). (4)$$

First k device evaluations are performed to get the nominal values. Then we need one function evaluation per circuit node (n) to get the displaced values for the partial derivatives. For each of these function evaluations we need k device evaluations.

However, to get each displaced circuit node value we only need to evaluate the components which are connected directly to the circuit node in question and to its neighbor circuit nodes. Thus, firstly we get n times

^{*}The neighbor devices contribute to the total current in the present node.

the cost for each circuit node. Secondly, for each circuit node we take all its neighbors, m-1, and evaluate all their device models, which are d each. This is obviously the worst case since some of these components generally are the same and these components only need to be evaluated once. In addition to these components we also need to evaluate the components which only contribute to the diagonal for the circuit node in question. The number of these "diagonal-only" elements per circuit node is called j, with  $j \leq d-m$ . Thus, we get a worst-case expression for the number of circuit evaluations in this node as

$$C_{col} = k + n((m-1)d + j)$$
 (5)

After insertion of the worst-case value for j and simplification we get

$$C_{col} \leq k + nm(d-1) \tag{6}$$

# Comparison of row-wise and column-wise schemes

As previously mentioned an electrical device only has a small number of terminals. Thus, a device typically has 1-3 ungrounded terminals, and consequently there are 2-3 three devices per circuit node. By letting the number of devices depend on the number of circuit nodes,  $k = \alpha n$ , we can compare  $C_{row}$  with  $C_{col}$ . Thus, by using the worst-case expressions for  $C_{row}$  and  $C_{col}$  we get

$$\frac{C_{col}}{C_{row}} = \frac{n((\alpha b)^2 + \alpha(1-b))}{n((\alpha b)^2 + \alpha b)} = 
= 1 - \frac{2b-1}{b(1+\alpha b)}.$$
(7)

As  $2b \gg 1$  we can simplify this further to

$$\frac{C_{col}}{C_{con}} \approx 1 - \frac{2}{1 + \alpha b}.$$
 (8)

Both coefficient evaluation schemes are now  $\mathcal{O}(n)$ . However, it is clear that the column-wise perturbation scheme still requires fewer device model evaluations, typically 70-85% of the number of row-wise evaluations. On the other hand, with the column-wise method, each device model evaluation is more complex as all terminal currents must be computed. With the row-wise scheme, only the currents for the terminal connected to the node in question need to be evaluated. Thus, the performance of both methods should be comparable.

Circuit		Row	Col	An
	#N-R	565	574	574
jc_two	#t-p	67	65	65
rank = 7	CPU (s)	6.6	6.9	5.1
	#N-R	619	619	649
two	#t-p	82	82	80
rank = 7	CPU (s)	24.4	21.7	15.2
	#N-R	649	649	643
jc_add	#t-p	76	76	75
rank = 105	CPU (s)	388	485	203
	#N-R	878	884	2016
$\mathbf{add}$	#t-p	120	128	136
rank = 106	CPU (s)	2449	2219	1899

Table 1: Jacobian coefficient computation method comparison

## Experimental comparison

The row-wise and column-wise perturbation methods for coefficient evaluation have been tested in CONCISE, where the comparison was done in a single computing node On a variety of circuits, both methods perform roughly the same, see table 1.

Differences between the methods in terms of Newton-Raphson iterations and time-points are due to the fact that the perturbation is computed row-wise and column-wise, yielding slightly different Jacobian matrices.

It is clear from the experimental data, that row-wise and column-wise coefficient evaluation perform roughly the same. The row-wise evaluation scheme has a great advantage in that the data structure is much simpler. With this method it is only necessary to know which devices are connected to each node, and not to keep track of neighbor nodes as in the column-wise scheme.

## 3 Stability problems

Because of symmetry in some device models, a difference approximation of the device derivatives sometimes yields a singular device Jacobian[†]. This is, for example, true for the MOS transistor. The MOS device is symmetrical with respect to drain and source. For an n-channel device the most positive, highest voltage, one

[†]Or near singular because of the limited numerical accuracy.

Circuit		Row	Col	An
fifo2 rank=136	#N-R	6758	7124	9551
	#t-p	2254	2403	1975
rank=136	CPU (s)	80000	106000	3730

Table 2: Convergence for the fifo2 circuit

Circuit		Row	Col	An
fifo2	#rej N-R	30156	45383	135
	#rej t-p	1212	1827	15

Table 3: Rejected time-steps for the fifo2 circuit

of the drain and the source terminals becomes the "de facto" drain terminal. Thus, if the drain to source voltage is smaller than the perturbation in the difference approximation, the source and drain of the device may change place during the derivative computation. When the terminals are changed, the same equations are used for both the source and the drain, yielding a singular device Jacobian. In the scalar case, when the rank of the Jacobian is 1 and only one device terminal is evaluated, the symmetry of the device causes no problem.

This stability problem has been experimentally verified on a CMOS circuit. In table 2 results are shown from a run with a test circuit causing sever problems for both perturbation methods. The number of Newton-Raphson iterations and time-points are comparable, but the CPU-times differ.

In table 2, the problem only shows up in the CPU-time column. By tabulating the rejected number of Newton-Raphson iterations and time-points, table 3, and comparing these with the number of accepted iterations and steps, the stability problems are more obvious.

It is interesting to note that, in spite of the problems with a near singular Jacobian, the waveform relaxation still converged, if slowly.

## Analytical derivatives

Analytical derivatives require the fewest number of device model evaluations, but make device equations more complex and the unpacking of data more difficult. The number of parallelizable operations is also smaller compared to the perturbation methods. With analytical derivatives, a device is evaluated only once per iteration. With row-wise or column-wise computation, a

CPU (s)		Number of coeff. eval.				
Circuit	rank	1	2	4	8	
add	106	4500	2380	1300	1140	
bufhot	8	80.0	45.0	35.7	37.8	
dflip	19	280	147	83.1	83.5	
jc_add	105	1540	981	982	987	
jc_dflip	19	40.6	28.3	28.8	29.8	
jc_ram	122	1460	1150	1160	1160	
jc_two	7	33.5	27.2	28.2	30.1	

Table 4: Run time with distributed Jacobian computation

device is evaluated once for each entry in the Jacobian, and once for each entry in the right hand side of the equation system. Thus, more, and smaller, tasks are available for parallel evaluation with the perturbation methods.

In the above tables a column for analytical derivatives has been added. Although the number of device evaluations is smaller than for the other methods, the required CPU-time is not much less. This is due to the fact that one device evaluation is much more complex than with the other schemes. In fact, for the MOS device the code is 5-10 times longer with analytical derivatives. Thus numerical derivation would be preferable if the stability problems mentioned above could be avoided.

A second problem with an analytical derivative computation is the fact that the results of the device evaluations have to be communicated in messages containing the entire device Jacobian. This Jacobian can have a rank from one to four for electrical circuits, and much higher for chemical "devices" [5]. Since the packing and unpacking of this data adds to the sequential fraction [3] of the subsystem evaluation, this scheme appears to be less attractive than, especially, the row-wise perturbation method.

# 4 Experimental results and further work

Row-wise difference approximation of the subsystem Jacobian matrix has been implemented in CONCISE. This version does not include the entire hierarchical waveform relaxation as depicted in figure 3 as we cannot partition circuits hierarchically presently. The test program is however ready to perform a hierarchical simulation once the partitioning part is ready.

[‡] Given that the stability problems of the perturbation method can be eliminated.

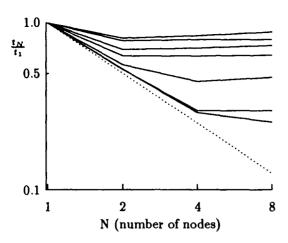


Figure 4: Inverted speedup with distributed Jacobian computation

CPU (s)	CPU (s)	Nun	nber of	coeff. e	val.
Circuit	rank	1	2	4	8
add	106	3680	1940	1090	810
bufhot	8	60.0	38.6	31.3	32.2
dflip	19	274	157	95.1	71.2
ram	123	2480	1350	854	667
two	7	24.3	15.3	12.0	1
jc_add	105	963	579	406	333
jc_dflip	19	25.9	16.6	13.9	14.2
jc_ram	122	860	539	434	440
jc_two	7	19.7	14.5	14.4	1

1 No result as rank < number of nodes

Table 5: Run time with distributed integration algorithm and Jacobian computation

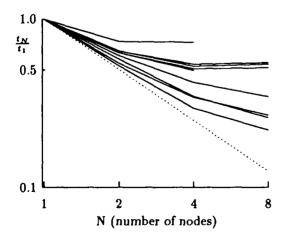


Figure 5: Inverted speedup with distributed integration algorithm and Jacobian computation

Results of runs on a Symult s2010 are shown in table 4. The table shows the CPU-time for various test circuits when 1,2,4, and, 8 computing nodes have been used for evaluating the Jacobian matrix.

The data in table 4 and figure 4 shows that a speedup of four can be achieved. This result is with accurate device models (complex MOS model and diffusion diodes), with very simple models (simple MOS model and linear capacitors, circuits with jc_ prefix) the speedup is limited to less than a factor of two. This confirms predictions in, e.g. [3].

The speedup does not depend strongly on the rank of the Jacobian, but more on what device models are being used. Thus, the speedup is limited by coefficient unpacking, matrix inversion, and the numerical integration in this case.

To verify this, the node voltage prediction, companion source computation, and integration error estimation was also distributed. These computations were spread out evenly over all the evaluation nodes. Thus a node program is responsible for solving the matrix equation, sending tasks to the device model evaluation nodes, and for controlling the integration algorithm.

With this approach data distribution is rather simple, but all nodes need to exchange node voltages and companion source values. The local truncation error estimations are sent to the computing node responsible for the integration error control and step length computation. A more efficient Jacobian matrix unpacking scheme, using precomputed pointers rather than search methods was also employed.

Somewhat better speedups are achieved by these modifications as can be seen in table 5 and figure 5. With the dflip circuit, some 15% of the total CPU-time of the node program was spent doing matrix inversions and another 15% to compute the Newton-Raphson iteration errors. Approximately 30% of the CPU-time was spent on packing and unpacking data. At this stage it seems to be more important to minimize the amount of data that has to be sent around than to distribute the LU-factorization.

Our results show that it is possible to combine a distributed direct ODE-solver and the waveform relaxation method. The speedup from the subsystem solver is not as good as for the waveform relaxation part. Thus, although being useful, the direct method is final solution to the waveform relaxation load imbalance problem. Dynamic partitioning is probably needed to further enhance performance.

## References

- [1] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits," *IEEE Transactions of Computer-*Aided Design of Integrated Circuits and Systems, vol. CAD-1, pp. 131-145, July 1982.
- [2] S. Mattisson, CONCISE a Concurrent Circuit Simulation Program. PhD thesis, Department of Applied Electronics, Lund University, Lund, 1986.
- [3] S. Mattisson, L. Peterson, A. Skjellum, and C. L. Seitz, "Circuit Simulation on a Hypercube," in The Fourth Conference on Hypercube Concurrent Computers and Applications, March 1989.
- [4] L. Peterson and S. Mattisson, "Partitioning Tradeoffs for Waveform Relaxation in Transient Analysis Circuit Simulation," in *The Fifth Distributed Mem*ory Computing Conference, April 1990.
- [5] A. Skjellum, M. Morari, and S. Mattisson, "Waveform Relaxation for Concurrent Dynamic Simulation of Distillation Columns," in Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, ACM, January 1988.
- [6] J. Vlach and K. Singhal, Computer Methods for Circuit Analysis and Design. New York: Van Nostrand Reinhold, 1983.